View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarWorks@UMass Amherst

University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Open Access Dissertations

2-2010

Generalized Instruction Selector Generation: The Automatic Construction of Instruction Selectors from Descriptions of Compiler Internal Forms and Target Machines

Timothy David Richards
University of Massachusetts Amherst, richards@cs.umass.edu

Follow this and additional works at: https://scholarworks.umass.edu/open_access_dissertations

 Part of the Computer Sciences Commons

Recommended Citation

Richards, Timothy David, "Generalized Instruction Selector Generation: The Automatic Construction of Instruction Selectors from Descriptions of Compiler Internal Forms and Target Machines" (2010). *Open Access Dissertations*. 178.
https://scholarworks.umass.edu/open_access_dissertations/178

This Open Access Dissertation is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

GENERALIZED INSTRUCTION SELECTOR GENERATION: THE AUTOMATIC CONSTRUCTION OF INSTRUCTION SELECTORS FROM DESCRIPTIONS OF COMPILER INTERNAL FORMS AND TARGET MACHINES

A Dissertation Presented

by

TIMOTHY D. RICHARDS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2010

Department of Computer Science

© 2010 Timothy D. Richards

GENERALIZED INSTRUCTION SELECTOR GENERATION: THE AUTOMATIC CONSTRUCTION OF INSTRUCTION SELECTORS FROM DESCRIPTIONS OF COMPILER INTERNAL FORMS AND TARGET MACHINES

A Dissertation Presented

by

TIMOTHY D. RICHARDS

Approved as to style and content by:

J. Eliot B. Moss, Chair

Charles C. Weems, Member

David Barrington, Member

Maciej Ciesielski, Member

Andrew Barto, Department Chair
Department of Computer Science

To Jody, Caleb, Hazel, Pixel, Andy, and Lumpy—I love you

EPIGRAPH

“Errors, like straws, upon the surface flow; He who would search for pearls must dive below....”,

-John Dryden

ACKNOWLEDGMENTS

Eliot Moss and Chip Weems have been excellent advisors during my graduate career. They dedicated a significant amount of their own time to provide me with the best possible graduate experience. Their complementary style has had a profound impact on my success during this time and has allowed me to reach this point in my career. They continually offer advice that helps form my ideas and research goals—for this, I am eternally grateful. I would also like to thank my thesis committee members David Barrington and Maciej Ciesielski for their feedback and time during this process.

I also acknowledge the members of the Architecture and Language Implementation Laboratory (ALI) for providing a fun and interactive environment well suited for generating interesting ideas and solutions. I am especially grateful to the rest of the CoGenT team, Trek Palmer and Ed Walters, for their hard work, dedication, support, and help to make GIST a reality. I also thank the undergraduate students involved as part of the Research Experience for Undergraduates (REU) program that spent their summers working on CoGenT and GIST. A special thanks to Addison Mayberry, Adam Fidel, and Chujiao Ma for their hard work in generating a significant amount of GISTs results during the 2009 summer program.

Finally, I am forever thankful to my wife, Jody Bird, for her commitment to this process and for getting me on track in 1994. Her support and encouragement has been vital and I could not have accomplished this without her smiles and laughter. I would like to thank my son Caleb for teaching me that ones imagination has no limits and pulling me away from work to help on the construction site is more important than I can begin to describe. I thank my daughter Hazel for arriving precisely when I could use those extra smiles and gentle giggles that ground me in this life. I also thank my friends and family, especially Nina,

for their words (and cards) of encouragement over the past several years, to popie/grampi for teaching me how to sweat pipes, and to the entire Bird family for all their concern, encouragement, phone calls, and laughter. A special thanks goes to Sam and Dana Tomlin for *everything* they have done to help my family during this time—I am always and forever, in Caleb's words, "ophay".

ABSTRACT

GENERALIZED INSTRUCTION SELECTOR GENERATION: THE AUTOMATIC CONSTRUCTION OF INSTRUCTION SELECTORS FROM DESCRIPTIONS OF COMPILER INTERNAL FORMS AND TARGET MACHINES

FEBRUARY 2010

TIMOTHY D. RICHARDS

B.A., CLARK UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

One of the most difficult tasks a compiler writer faces is the construction of the instruction selector. The instruction selector is that part of the compiler that translates compiler intermediate representation (IR) into instructions for a target machine. Unfortunately, implementing an instruction selector "by hand" is a difficult, time consuming, and error prone task. The details of both the IR and target instruction set must be carefully considered in order to generate correct and efficient code. This, in turn, requires an expert in both the compiler internals as well as the target machine. Even an expert, however, can implement an instruction selector that is difficult to verify and debug. In this dissertation we describe the instruction selector problem, cover previous attempts at solving it, and identify what we believe to be the most prominent factor inhibiting their widespread adoption.

This dissertation proposes a generalized approach toward generating instruction selectors automatically. In particular, we propose CISL, a common machine description language for specifying the semantics of compiler IR and target instructions, and GIST, a machine independent heuristic search procedure that can find equivalent instruction sequences between compiler IR and target instructions. CISL is an object-oriented-based language leveraging modern programming language constructs (e.g., classes, inheritance, mixins) and is capable of describing instructions for a variety of IR and target ISAs (Instruction Set Architecture). GIST leverages CISLs well-defined semantics and a canonicalization process to discover automatically instruction selector patterns: target instruction sequences that implement IR semantics. These instruction selector patterns are then generated in a compiler implementation independent format (XML). Small adapter programs use the generated instruction selector patterns to generate compiler specific implementation code. Our experiments show that instruction selector patterns can be discovered automatically and independent of a particular compiler framework or target machine. In addition, experience proved that adapter programs are easy to implement and instruction selector code is easy to generate from generated patterns. Furthermore, the generated instruction selectors are comparable in performance to the original compilers.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

In this work we show that the instruction selector phase of a compiler can be generated automatically from machine specifications describing a compiler intermediate representation (IR), target machine, and memory implementation mapping in a manner that is independent of the source compiler framework and target architecture. This work is motivated by a recognition of the importance of compilers in the future of computing and their automatic construction [Hall et al., 2009], the lack of a compiler- and architecture-independent solution to the problem, and an appreciation of the difficulty of the problem when it is generalized.

The ubiquity of software systems in today's society is unprecedented. This phenomenon can, in part, be attributed to the continued increase in microprocessor performance as predicted by Gordon Moore in 1965 [Moore, 2000]. At the same time, faster processors allow larger and more complex software systems to be developed that are capable of solving a wider range of problems that were previously considered impossible including the design of faster processors.

This complexity is caused by the many layers of abstraction that exist between the user of a computer application and the actual hardware that is executing the software. For example, a typical user may interact with a Java-based application that has been developed using a number of independent libraries, all having their own data structures, algorithms, and *application programming interface* (API), interpreted by the Java Virtual Machine (JVM), which is natively executed by an underlying microprocessor such as a PowerPC, SPARC, or

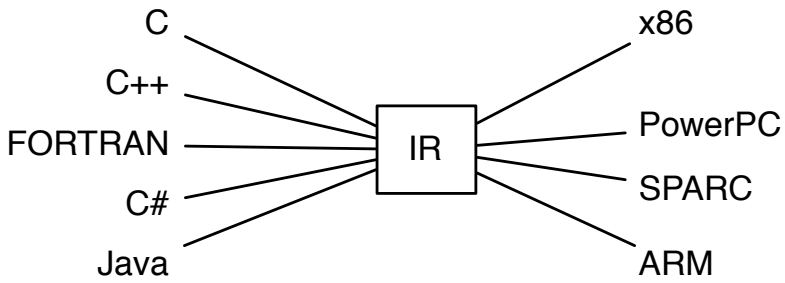


Figure 1.1: IR Interface

x86 processor. Furthermore, each processor may in turn have several levels of abstraction including its *instruction set architecture* (ISA), *micro-operations*, and cache and pipeline semantics.

At the heart of all these complex layers of abstraction lies an important computer program making much of this possible: the *compiler*. The compiler, as originally conceived by Rear Admiral Grace Murray Hopper and made successful by the first FORTRAN I compiler, allows high level programming languages (HPL) to be translated into instructions that can be directly executed by a specific target machine. Compilers free programmers from the difficulties of writing assembly programs, which require intimate knowledge of the underlying computer architecture, are difficult to manage in the large, and are difficult to port. Thus, the compiler is one of the critical links in computing history upon which all modern software and the HPLs used to create them is based. It is the compiler that has allowed software to flourish and become a necessity of modern day society.

Although the compiler removes the burden of machine details from the shoulders of most software development, compilers themselves are not as fortunate, as they are by their very nature machine-specific. To solve this problem, much effort has gone into designing compilers for retargetability (i.e., compilers that can be more easily modified to produce code fr new architectures). One way to accomplish this is by dividing compilation into well defined phases where each phase focuses on a specific task. This allows phases that are

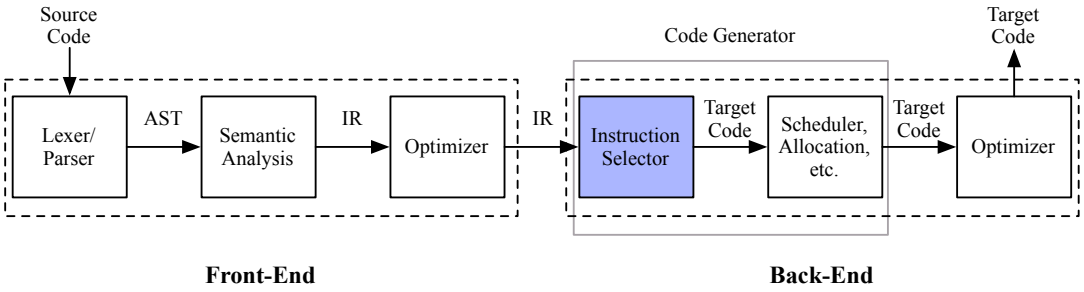


Figure 1.2: Retargetable Compiler Phases

machine-independent to remain the same across architectures and those that are machine-specific to be replaced more easily.

At some point, however, the machine-independent part of the compiler, also known as the *front-end*, must interface with the machine-dependent part of the compiler (the *back-end*). This is accomplished using a compiler's *intermediate representation* (IR). The IR is an instruction set for an abstract machine that allows the programming language semantics to be captured exactly while still remaining independent of any particular architecture. The benefits of such a representation are two-fold: low-level optimizations such as dead code elimination and branch-to-branch removal can occur independently of the target, and more importantly the translation into actual target instructions can occur independently of the programming language being compiled, as depicted in Figure 1.1.

In Figure 1.2 we highlight the major phases of a retargetable compiler. The *code generator* (CG) is that part of the compiler that is responsible for translating front-end generated IR into target machine instructions. The CG itself is divided into a number of phases including *register allocation*, *instruction scheduling*, and *instruction selection*. Our emphasis (and that of prior work) is on the instruction selector. However, we use the term "code generator" interchangeably in this dissertation and make a distinction only when necessary. A code generator is implemented by carefully mapping an IR sequence to a target instruction or sequence of instructions. Ideally, when the code generator is presented with a sequence of IR instructions it will produce a sequence of efficient target instructions that correctly

implement the IR semantics. As it turns out, building *good* code generators "by hand" is difficult to achieve in practice without error.

Constructing a good code generator for modern compilers and modern architectures is a formidable task. A good code generator must implement programming language constructs in the target instruction set correctly and it must utilize the most efficient instructions. Efficiency is often measured in terms of execution time or space but can also refer to aspects such as power usage. For example, knowing how many machine cycles a particular instruction requires to execute would allow us to choose instructions that are faster. For variable length instruction encodings (e.g., x86), replacing a longer sequence of instructions with a single instruction could result in a reduction in the overall number of instructions stored in memory, which could result in a reduction in cache misses and page faults. Thus, to build a code generator that is both correct and efficient requires substantial knowledge of the programming language internals, in particular the compiler IR, as well as a mastery of the target architecture.

In addition to the complexities of compiler internals and the intricacies of the target machine, software of this sort must consider efficiency. This concern is becoming increasingly important in systems that employ *just-in-time* compilation strategies such as the Java Virtual Machine (JVM) and the Common Language Runtime (CLR). In this scenario the programming language (i.e., Java or C#) is translated into an IR, typically referred to as *bytecode*, that is incrementally compiled as the program is executed by a virtual machine. The enormity of information required by compiler writers and the complexity of these systems lead to longer development times, code generators that are error prone and potentially inefficient, and compilers that are ultimately difficult to debug.

Given the importance of code generators in today's world, is it possible to improve the current state of compiler back-end construction? In particular, is it possible to reduce the amount of time required to build a code generator such that a compiler can target a new architecture with ease? One might propose that we use *brute force* methods and increase

the number of persons developing a particular CG, hoping that in the end that might reduce overall time and increase quality. Unfortunately, the enormity of this kind of software is not attributed to the vast number of modules or lines of code but rather to the immense knowledge base from which the individual must draw in order to implement a few lines of code that correctly implement the semantics of the source language as instructions of the target machine.[1]

If increasing the number of persons building a code generator is not a viable option, perhaps the solution to this problem is automating the construction of a CG. In particular, can we gather the information that is typically required by the implementor in such a way that it can be processed by a computer program to generate a code generator? Furthermore, can this program build a code generator for *any* compiler that can target *any* machine? Can the construction of a code generator be automated similarly to the lexer and parser components that are generated by lex and yacc?

If we can generate a CG automatically, is there sufficient reason to do so? Over the past decade the diversity of general purpose processors has decreased. Two notable examples include the phase-out of the Alpha microprocessor (originally developed by Digital Equipment Corporation (DEC)) in early 2001 (in favor or the x86) after Digital was purchased by Compaq, and the switch from PowerPC chips to the Intel x86 processor by Apple Computer in June 2005. This would indicate a movement toward a single general purpose CPU and less need for generating instruction selectors automatically. If we look toward the future, however, it is becoming clear that Moore's law is coming to an end and computer architecture is at a crossroads. Increasing clock frequency is no longer an option for increasing performance of next-generation machines because of a myriad of physical limitations such as power consumption, heat dissipation, and wire delay.

Because of these problems heterogeneous multicore architectures and machines with extended instruction sets are becoming more prevalent. Building instruction selectors

[1] Of course, as Frederick Brooks clearly points out [Brooks, 1995], one should never expect to increase software development productivity by throwing person power at the problem.

quickly for these new architectures, such as the Cell Broadband Engine [Kahle et al., 2005, Gschwind, 2006], will become increasingly important to allow experimentation to proceed more rapidly. In addition, people are expecting more from their portable devices (e.g., cell phones, hand-helds, netbooks). This has led to a larger diversity of chips in embedded systems, including general purpose CPUs, Harvard architectures, VLIW, DSP, ASIC, and FPGA. This architecture zoo demands a new level of automation and flexibility for compilers, especially the instruction selector, to build next-generation devices and deliver at a pace that people have come to expect.

The questions of this dissertation have been asked since the advent of the compiler. Most answers have manifested themselves in solutions involving the construction of a program, commonly referred to as a *code generator generator* (CGG) or instruction selector generator (ISG), that attempts to generate automatically the instruction selector component of the compiler. Although previous proposals have provided great insight into the complexities of the problem most have fallen short of delivering a *general* solution.

A general solution is one that maintains a careful balance between generality, automation, and compiler design flexibility. Prior work emphasized automation over generality. For example, previous systems often assume several aspects of the code generator such as a specific IR, memory model, output (e.g., assembly, machine code), register allocator, or instruction scheduler. Although this may lead to a more "automatic" solution, it is typically not possible to use the CGG results with other compiler frameworks or architectures. In addition, it restricts a compiler writer's ability to explore different implementations of the IR semantics on the target. This dissertation introduces a general strategy for generating an instruction selector automatically from a description of the compiler IR, target instruction set, and memory mapping specification. To evaluate this approach we developed GIST, an implemetation of the generalized approach to generating instruction selectors automatically and independently of any particular compiler framework and target architecture. We demonstrate the generality of GIST using multiple target architectures (PowerPC, MIPS,

Sparc, x86, and ARM) and two very different frameworks: `lcc` BURS rule patterns and Jikes RVM Baseline compiler expansion of Java bytecode to native code. We show the effectiveness of the generated selector by passing its output to existing frameworks through small adapters and comparing the resulting compiler outputs against their standard versions.

1.2 Posing the Problem

Compiler writers have a broad suite of tools to assist with important front-end compiler tasks such as scanning and parsing, but comparatively fewer for dealing with back-end tasks such as instruction selection and code generation. Those that do exist assume a fixed compiler framework, i.e., they are dependent on a source or target representation associated with a single compiler intermediate representation (IR) or virtual machine (VM) specification. At the same time, the ongoing move to greater parallelism is requiring compiler writers to experiment with novel optimizations that involve IR or VM extensions, and to deal with a proliferation of instruction set architecture (ISA) extensions. To be widely applied and evaluated, such work should not be confined to a specific compiler framework or a limited set of target architectures. The alternative to using framework-specific back-end tools is to construct these components by hand. But that process is time consuming and error-prone, which impedes the flexibility of experimentation that is critical to new discoveries.

For example, say a research team is designing a novel programming language that will target a new VM that has rich run-time semantics. Multiple iterations of experiments with compiler optimizations, IR semantics, and tuning the VM design will necessitate reworking the back end. Similarly, testing transactional memory extensions for a current processor through simulation depends on corresponding adaptations in the instruction selection and code generation compiler phases. There are many examples in which functionality moves back and forth across the hardware-software interface during these kinds of research efforts,

before the correct balance can be achieved. But when the cost of modifying that portion of the compiler is high, researchers are deterred from broader exploration and evaluation.

As mentioned previously, one of the main difficulties in automating back-end generation is generating instruction selectors. To appreciate why this is hard, consider the challenge of automatically deriving an instruction selector that maps the instructions in a stack-oriented IR such as Java bytecode, to a load-store architecture. The semantic gap between these representations may seem easy to bridge to achieve basic functionality for a single instance, but doing so automatically in a manner that takes advantage of modest experimental variations requires extensive analysis of the two semantic domains. Prior instruction selector generators simplify the problem by fixing the IR and allowing variation only in the target, which greatly reduces the search space for semantically equivalent instruction sequences. But this approach deals only with extensions to the target architecture and fails to enable researchers to experiment with novel IR extensions and language features that depend on them. Some instruction selector generators further simplify the problem by making assumptions about the characteristics of the target architectures or the rest of the back end.

GIST is the first instruction selector generator that is tied neither to a specific compiler framework nor to the type of target. As a result of removing these assumptions we were forced to deal with a much more general instruction selector generation problem. Our solution required us to develop the means of describing a wider range of IRs, VMs, and ISAs, to handle matching of much larger IR graphs, and to apply more search optimizations to reduce the work involved in exploring the exponential growth in potential matches.

We present here the key ideas behind the GIST system by walking through the process of using it. GIST takes as its primary inputs descriptions of a compiler IR and a target, and produces a mapping between the two representations, in the form of instruction selector patterns, via canonicalization and heuristic search. GIST outputs the patterns in a compiler framework independent XML form. One then writes a small adapter that converts the XML

into code, tables, etc., suitable to plug in to the host compiler framework. Because of the transparency and regularity of the XML representation, we have found these adapters to be small and simple to write—in fact, several of our adapters were written by undergraduate students with no prior compiler-writing experience, while working in our lab during an eight-week summer program.

GIST is highly configurable and facilitates rapid prototyping of experiments. For example, as a secondary input GIST takes a user-specified store mapping that enables a designer to specify different choices in the correspondence between the resources of the IR and target (e.g., how much of the stack to keep in registers versus memory). GIST also uses a database of semantic axioms, which the user can extend in order to assist GIST in mapping between operations where semantic equivalence depends on an orthogonal rule, such as splitting long arithmetic into multiple cases of standard length arithmetic, or breaking large constants into pieces small enough to fit in the immediate fields of instructions.

The contributions we offer in this dissertation include:

- The design and implementation of the CISL [Moss et al., 2005] description language supporting a wide range of target ISAs, compiler IRs, and VM specifications while maintaining its independence from existing frameworks.
- Developing the means for describing the mapping between the resources of the source and target. We do this separately from the source and target semantics because it embodies an important set of design choices over which a compiler writer must have control, e.g., how can we map a source stack abstraction onto a target memory? Keeping store mappings separate from the source and target description also avoids polluting them with semantic information that should not be directly tied to either.
- Adding the capability to express axioms that extend GIST's knowledge of semantic equivalence, which already includes associativity and commutativity of arithmetic and boolean operators, etc. We note that in a separate project we have developed

the means to prove such axioms correct using satisfiability solvers, so we need not simply trust the user to get axioms right.

- Solving the generalized instruction selector generator matching problem, which involved choosing appropriate algorithms and heuristics.
- Minimizing the effort required to adapt the GIST output to an existing framework. While the descriptions and search process for GIST are framework-independent, at some point the generated instruction selector must interface with a code generator and front end for the results to be useful. We have worked to make creating adapters both simple and easy, and indeed we have shown that writing them does not require extensive experience with compilers, and that they are at least an order of magnitude smaller than ISA descriptions.
- Evaluating GIST in the context of two very different compiler frameworks and multiple target architectures, including generation of instruction selectors for targets that were not previously supported by the frameworks.
- Validating GIST for those targets where code generators are available. We show that our generated instruction selectors produce code that has performance essentially identical to that of the existing selectors.

The remainder of this chapter proceeds as follows: In Section 1.3 we provide a definition of what a valid CGG solution is, which will allow for a point of comparison between our research versus prior work in this area. Section 1.4 discusses the big picture and where this work resides in a larger context. Finally, Section 1.5 provides an overview of what GIST does and how it works.

1.3 Prequisites for Valid CGG Solutions

The problem of automatically generating efficient code generators for compilers has a long history—yet the solutions have resisted widespread adoption. There are several factors that

contribute to this lack of acceptance. The most obvious issue is code generator generators that either *do not exist* or *do not work*. Without a working implementation it is hard to prove or disprove the proposed solution. If we leave ourselves with the set of proposals that have working implementations we must consider three important criteria that contribute towards a valid CGG solution: *correctness*, *efficiency*, and *generality*.

A CGG must not only have a working implementation but must also guarantee that the target instructions produced from a given sequence of compiler IR are correct. In other words, we must be able to prove that the semantics of the compiler IR and the target instructions are identical at an instruction by instruction basis, as well as across sequences of instructions.

Given a correct CGG implementation, we must then consider three *criteria of CGG efficiency*: the efficiency of the CGG, the efficiency of the actual code generator, and the efficiency of the code generated by the code generator. The length of time it takes a CGG to produce a code generator must remain within *acceptable* limits. In this thesis, we define acceptable as the amount of time it takes the CGG to produce a working code generator such that it can be used for building real compilers. For example, A CGG taking months to produce a code generator is hardly useful compared to a system that completes in a few hours especially when time is precious, as usual.

In addition to the cost of executing the CGG we must also be concerned with the efficiency of the generated code generator. This is becoming increasingly important today with the use of JIT-style compilation systems such as for the Java Virtual Machine (JVM). Just as the significant overhead of garbage collection algorithms cast a shadow on the LISP systems of the 1970s, we must give particular attention to the cost of online code generation systems. That is, the cost of executing a JIT compiler must not dominate the overall time of the executing program.

Although correctness and efficiency are paramount in a working CGG system, an often overlooked requirement is the practicality and generality of the system. The CGG must be

practical in that the work required to run the system must be less than the work required to implement a code generator for a particular compiler system "by hand", and it must be general in that it can be used for many architectures and compiler systems. In particular, the method used to generate a code generator must not only be independent of the target machine and its instruction set (i.e., RISC vs. CISC) but must also be independent of the compiler infrastructure and its IR that will use the code generator (a limiting factor that has plagued prior CGG solutions).

In summary, for a CGG to be considered a valid solution it must have a working implementation that is compiler- and architecture-independent. The implementation must adhere to our three criteria for efficient CGG: efficiency of the code generator generator, the generated code generator, and the code generated by the code generator. Lastly, the CGG must be practical in that the overall work required to use the CGG must be less than the work required to craft a code generator by hand. We use these criteria to validate the GIST system.

1.4 Background

CoGenT stands for Co-Generation of Tools, particularly compilers and simulators. Compiler and simulator tools for systems research are difficult to develop and coordinate since each tool is complex in its own right, and both are dependent on aspects of the target architecture. The CoGenT project, illustrated in Figure 1.3, addresses this problem by providing two sets of components: multiple coordinated specifications that describe the target instruction set architecture, compiler IR, and micro-architecture; and a set of tools that process these specifications and produce compiler and simulator components such as code generators, instruction schedulers, and simulators at varying degrees of detail. Generating a compiler and simulator from the same set of descriptions ensures that the two will always be consistent, and since automatic tool generation is fast, CoGenT allows designers to explore more design space in less time.

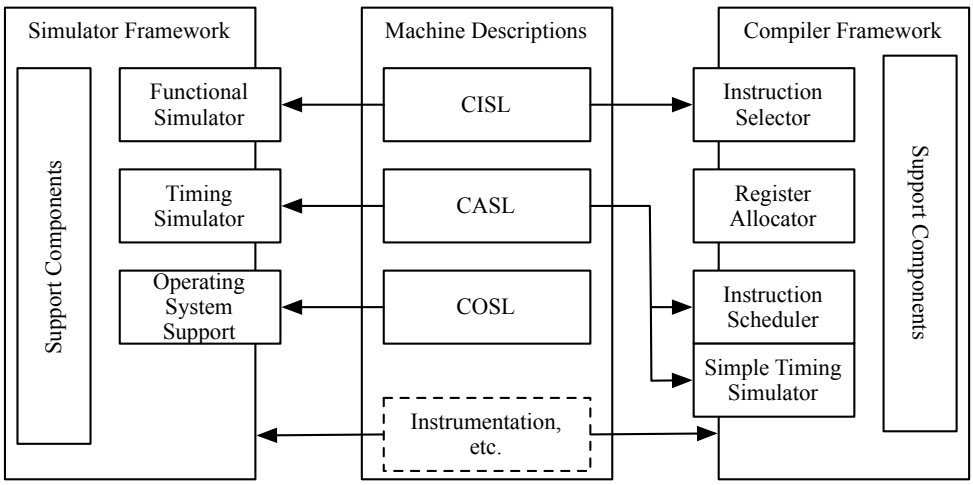


Figure 1.3: CoGenT Project

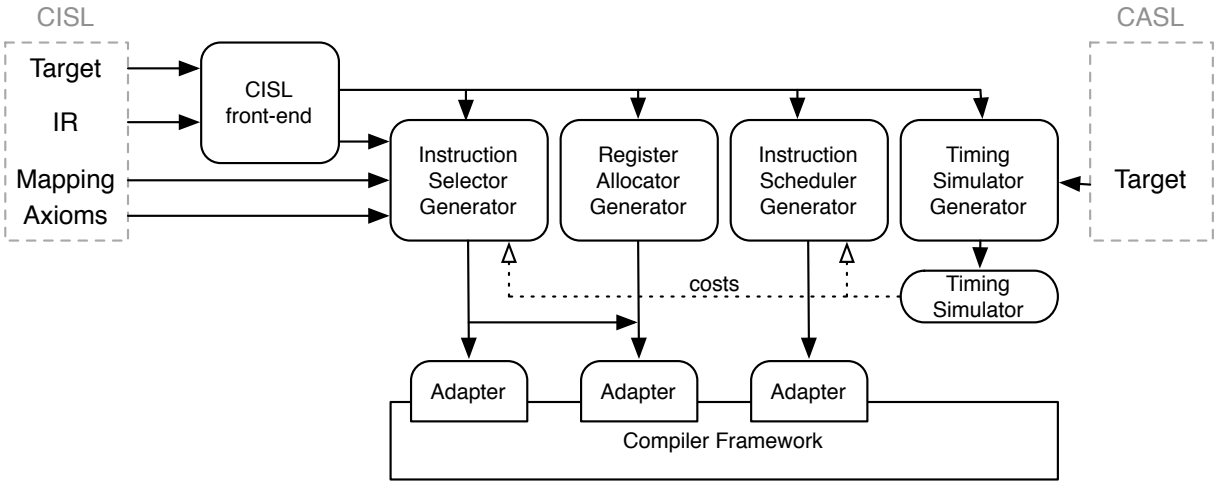


Figure 1.4: CoGenT Compiler Generation

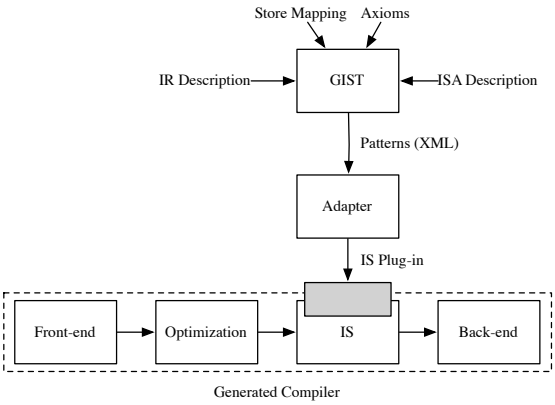


Figure 1.5: Role of GIST Compiler Construction

Prior systems tend to use either a simple language, making descriptions unwieldy, or a machine description language augmented with a complete general purpose language, rendering analysis difficult. We address this issue by providing multiple coordinated languages. Currently, CoGenT has three primary description languages:

1.5 Overview of GIST

GIST is designed to be used anywhere that a translation between two instruction representations is needed, e.g., binary translators, interpreters, etc., but its primary use is in compilers. Figure 1.5 shows the relationship between GIST and a compiler. It is important to note in this figure that GIST is not a compiler phase; rather, it *generates* a compiler phase. In particular, it is necessary to keep in mind that the heuristic search we describe later happens when generating the compiler, *not* as the compiler runs.

Here we summarize the process one follows in using GIST:

1. Define the source and target of the translation process. In a typical case, the source will be a compiler intermediate representation (IR) or a virtual machine (VM) specification, and the target will be an instruction set architecture (ISA) or another virtual machine. Figure 1.6 shows a variety of other combinations for which GIST would

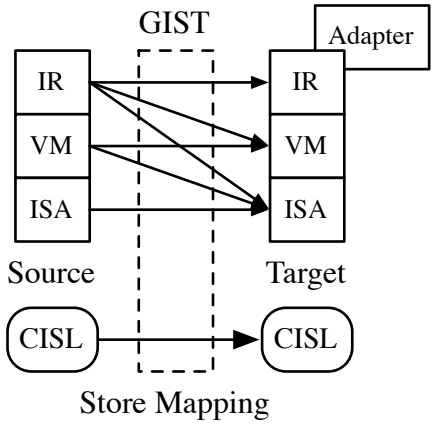


Figure 1.6: Scope of GIST

be appropriate. For example, for a binary translator both source and target could be ISAs, or a compiler framework could be targeted to a virtual machine. Typical uses of GIST are situations where the target representation is at an equal or lower level than the source, although the reverse may be possible and useful.

2. Create descriptions in the CISL description language for the semantics of both the source and the target. While CISL can describe both syntax and semantics, only the semantics are strictly necessary for matching purposes. Syntax is useful in other tools that one can generate from CISL, such as assemblers, disassemblers, and functional simulators. CISL descriptions are the primary inputs to GIST. We elaborate on their form in Chapter 3.

3. Specify how to map source storage elements to the target. This is an important aspect of the framework that the compiler designer wants to control in experimenting with different optimization options. For example, one can describe the layout of Java arrays, which affects how instructions are selected for fetching the size field when performing bounds checks, and for computing the offsets of elements for array accesses. This is the only input that the GIST user will write for each source and target

pair—the other descriptions are independent of the pairing, and can be reused in any combination. In fact, many users will want to write multiple mappings that encapsulate different design decisions. For instance, one could use the store mapping to vary the dispatch mechanism in a virtual machine and set up a head-to-head comparison. We further describe the mappings in Chapter 6.

4. Add any algebraic axioms that will assist matching in cases where additional independent semantics are helpful. For example, one useful axiom breaks 32-bit integers into two 16-bit halves to make it more straightforward for an ISA to manipulate them in certain ways. Axioms tend to be useful in many contexts, and thus are frequently reused, once written. We anticipate that each new target, and possibly some sources, may need a small number of new axioms to achieve the best patterns.

5. (Optional:) Specify how GIST should prioritize among multiple valid output code sequences. For example, one could choose the sequence with the fewest instructions, or with the least number of bytes, or, if one supplies a speed or power estimator, the sequence with the fastest speed or least power consumption. By default GIST selects the sequence with fewest instruction, but it allows the user to plug in a custom evaluation function for assigning a quality value to instruction sequences. We describe some of the alternatives in Chapter 6.

6. Execute GIST. It goes through its search process and outputs XML patterns. The user must determine if the coverage and quality of the patterns is acceptable, using the detailed feedback supplied from the search process. If the results are not acceptable, then the user iterates through the previous steps as necessary.

7. Write a small *adapter* that turns the framework independent XML instruction selector patterns into a plug-in that is suitable for inclusion in the instruction selector phase of the specific compiler framework one is targeting (see Figure 1.5). The tar-

geted phase typically comes between the optimization and code generation (register allocation/instruction scheduling) phases.

It is important to note that we have already developed descriptions for two popular compiler framework IRs (LCC and JVM), a partial description of the PQCC compiler (for verifying prior work), and four widely used architectures (PowerPC, IA-32, Arm, MIPS), along with a number of other prototypes. Mappings exist for many IR-ISA pairings of these frameworks and ISAs, along with a large set of commonly useful axioms. We have also implemented several adapters. We continue steadily to expand our library of these components. Thus, most users will jump directly into modifying existing GIST inputs and adapters for experimentation, rather than having to write new descriptions from scratch.

It also bears repeating that the CISL language was designed so that its descriptions can be used to generate the additional components in the back end, plus related development tools such as assemblers, disassemblers, debuggers, and functional simulators. Thus, even when it is necessary to write a new CISL description, the benefits will eventually go well beyond producing instruction selectors.

CHAPTER 2

RELATED WORK

Much work has contributed towards the advancement of code generator generation. This chapter surveys past approaches and existing tools for code generation, code generator generation, and the artificial intelligence and theorem proving methods and practices that have influenced this area. The similarities and differences of GIST are highlighted with respect to prior work as each is presented.

2.1 Retargetability

A compiler that is retargetable is of much greater use than one that is not. Many compiler textbooks emphasize this fact by defining several distinct phases of the compiler [Hanson and Fraser, 1995, Muchnick, 1997, Aho et al., 1986, Fischer and Richard J. LeBlanc, 1991, Cooper and Torczon, 2004, Appel, 1998], most notably the front-end and back-end (code generator). The ability to generate code for multiple target architectures given a single source program and compiler suite greatly simplifies the job of the user of the compiler. More specifically, a compiler is more versatile when the target code it generates can be easily modified or changed. This is particularly advantageous in the area of architecture and compiler research. For example, experimental architectures require compilers that can easily adapt to changes in the instruction set and compiler researchers require code generators that can be modified seamlessly to investigate new target architectures.

A retargetable compiler is one that has been carefully designed to separate the machine-independent aspects from those that are machinedependent. This separation of concerns is accomplished primarily through the use of a virtual instruction set, typically referred to

as a compiler's *intermediate representation* (IR). IRs are designed to model machine-level constructs such as registers, main memory, and operations on them yet remain at the level of abstraction necessary to allow independence of any particular architecture. In other words, an IR acts as an abstract *interface* to any machine whereas the machine-dependent code generator is the implementation. The goal of GIST is to generate the implementation part (instruction selector) of such a retargetable compiler automatically, given the definition of this interface and the implementation language (i.e., target instruction set).

A prime example of the success of a retargetable compiler is the open source compiler GCC [Stallman]. GCC Version 3.3 can be hosted and generate code for well over a hundred different architecture/operating system combinations (approximately 51 different architectures) and possibly more if you include the many specialized derivatives. This myriad of supported programming languages and target architectures is the result of two important aspects of GCC: the number of developers actively working on various code generator ports and its loosely defined IR generically named *register transfer language* (RTL).

To create a new code generator for GCC involves the definition of a set of C macros (as GCC is written in C) that define machine specific details such as endianness and bit widths of the target as well as a LISP-like (S-expression) RTL pattern language that maps an RTL pattern to a target machine instruction(s). Code generation then becomes a matching problem on source RTL trees to target instructions to generate machine-specific assembly code.

Although GCC supports a wide range of target architectures it is not ideal for compiler exploration and architecture experimentation. Much of GCC's success is not so much because of an easy to use compiler infrastructure but rather through its large user and developer pool. A relatively large number of developers can work at supporting a new target and many users can work at discovering the bugs. This process continues until the quality of code produced is at an acceptable level. Unfortunately, this kind of development model is not feasible when a target instruction set is prone to changes. In other words, through

experimentation of the underlying machine, its instruction set, and compiler an acceptable level of quality might never be reached.

In addition, it is a single compiler framework that uses a specific implementation language (C). It supports a particular suite of source languages, missing several. Thus, if one wanted to explore a new architecture in the context of a programming language not supported by GCC, a new front-end would be necessary. Furthermore, it targets a particular run-time system which limits important research avenues (e.g., alternate styles of garbage collection). Lastly, its compilation strategy is fixed and only targets ahead-of-time compilation. For these reasons, GCC is not the ideal compiler framework to explore compilers and architectures.

A retargetable compiler is only a small part of what makes a compiler truly portable and useful as a research vehicle. In addition to adapting easily to the source programming language it must also be amenable to a changing target architecture. Furthermore, its reliability and quality must be invariant to modifications of that target machine. A clearly defined IR is important in achieving this goal, but how that IR is matched to the target instruction set is paramount. In the following sections we cover various code generation techniques, spanning manual techniques to more automated methods that are based on the idea of a retargetable compiler and its IR interface.

2.2 Manual Construction of Code Generators

A code generator for a typical compiler, at the most fundamental level, becomes a case analysis of an IR instruction to determine the appropriate target instruction that performs an identical computation. In particular, the IR instruction must be implemented by a target instruction based on the operator of the IR (such as addition or subtraction) as well as its operands. The operands of an IR instruction often refer to programmer-defined variables in the source language. For a register-based target these variables must be allocated and stored into a suitable location (i.e., register, memory) on the target. As such, where those variables

have been allocated on the target influences the possible choice of target instructions that can implement the IR instruction. Where these variables reside is determined by the *register allocator*, a back-end phase that can occur before or after instruction selection.

2.2.1 Macro Expansion

A basic code generator implemented "by hand" must carefully consider all the alternatives and manually write source code that *macro expand*s the IR into a sequence of target instructions based on these criteria. Consider the *three-address*[1] IR instruction `a = b + c` (where `a`, `b`, and `c` are variables declared in the original program text). The target instructions that we generate depend on where the variables have been allocated on the target. It is entirely possible that one or more of these variables reside in a target register if it has been recently used by a previous IR instruction, in the heap if dynamically allocated, or on the stack if allocated by the current frame. The following pseudo-code of the `emitAdd` procedure is an example of what might be manually coded to generate PowerPC instructions for the above IR,

[1] Three-address code is one kind of IR representation; we introduce others as they pertain to the approach being discussed.

Procedure `emitAdd(`*IRInstruction inst*`)`

Input: a = b + c

if *b is not in a register* **then**
    Allocate a register Rb for the value of b;

    Generate "lwz Rb," + gen_location(b);

end

if *c is not in a register* **then**
    Allocate a register Rc for the value of c;

    Generate "lwz Rc," + gen_location(c);

end

Allocate a register Ra for variable a if not already in register;

Generate "add Ra,Rb,Rc";

The procedure `emitAdd` determines if variables `b` and `c` are in registers and if they are not generates instructions to do so (i.e., the PowerPC lwz instruction). It uses the register allocation utility function `gen_location` to retrieve the exact location (possibly generating more code if the variable resides at some offset from a well known location such as the frame). It then allocates a register location for `a` (if `a` is not already in a register) and finally emits the `add` instruction.

2.2.2 Maximal Munch

Code generation can also be viewed as matching patterns on IR fragments represented as trees. For example, the IR instruction for `a = b + c` could be represented by the compiler in tree form as shown in Figure 2.1. Code generation then becomes the task of *tiling* the input IR tree with a minimal set of tree patterns. In addition, these tree patterns can have additional properties (such as cost factors reflecting time and space) that guide the code generator in its selection of the most appropriate pattern to apply. The algorithm for locally optimal tiling of an IR tree is called *Maximal Munch* [Cattell, 1980]. The algorithm

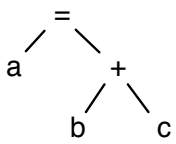


Figure 2.1: Maximal Munch Input IR Tree

is straightforward and begins at the root of the input tree and locates the largest tile/pattern that fits (breaking ties using cost factors). If a tiling does not exist, the algorithm is applied recursively to the subtrees. A tiled subtree is then replaced by the location of the result of that subtree. Once the subtrees have been tiled the root can then be tiled based on these results. If no tiling exists at this point, the IR cannot be tiled by the given patterns. Possible patterns that can tile the IR tree defined above are shown below in Figure 2.2.

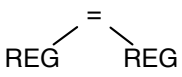
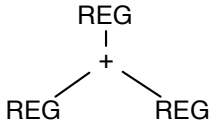
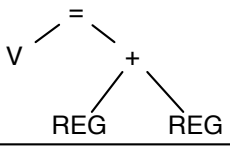
Emit	Patterns
RULE 1 lwz Ri, gen_location(V)	
RULE 2 addi Ri, gen_location(V), 0	
RULE 3 add Ri, Rj, Rk	
RULE 4 add Ri, Rj, Rk	

Figure 2.2: Maximal Munch Patterns

The patterns on the right show the possibilities of tiling a given IR tree and the PowerPC code on the left that is emitted by a successful match. A *V* indicates a variable and *REG* a register location. Given these patterns, it is possible to tile our input tree by applying

Rule 1 to both leaves containing `b` and `c` and then applying Rule 4 to the overall tree. This results in the following PowerPC code:

```
lwz Rj, location of b
lwz Rk, location of c
add Ri, Rj, Rk
```

Naturally, maximal munch works in unison with register allocation to determine where the current location of the variable might reside. In the above example, we are assuming that `b` and `c` are located in memory and `a` has already been allocated a register (in this case `Ri`). Additional code may be generated depending on the exact location of these variables. The implementation of maximal munch would require code representing these patterns and the tiling engine that would apply the patterns, traverse the IR tree, and manage the results of pattern application.

2.3 Automatic Generation by Pattern Specification

In the previous section we focused on manual techniques for constructing code generators. Here, we introduce techniques that have been proposed for generating a code generator automatically by specifying IR and target instruction patterns. These pattern specifications form rules that define how incoming IR sequences are matched and which target instructions are generated from them. In this sense, the emphasis here is on the generation of the implementation of the code generator rather than on determining automatically which target instructions implement which IR instructions. Prior work for the latter is presented in Section 2.4.

2.3.1 LR Parsing Techniques – Graham/Glanville

The success of lexer and parser generators spured automatic code generation techniques based on similar methods [Glanville, 1977, Glanville and Graham, 1978, Graham et al.,

1982, Aigrain et al., 1984], in particular, the application of LR parsing theory to code generation. In this approach, a grammar of the source IR is defined as a list of productions of the form $l \rightarrow rt$. The symbol l to the left of $\rightarrow$ designates the destination of a computation and the symbol r to the right specifies an IR instruction computation, in a linearized prefix representation, and a corresponding target instruction in assembly format (denoted by t). Prefix form is required to allow LR parsing to make decisions based on a limited-lookahead. For example, for the expression + e_1 e_2, where e_1 and e_2 are arbitrary expressions, a decision must be made based on the prefix of e_1 as e_1 can have arbitrary length. A left-hand side of λ indicates that there is no register result, that is, the instruction is executed only for its side effects (i.e., a memory store instruction). The following is an example of how we might specify such a grammar,

Rule 1	$r.1$	$\rightarrow$	$+\ r.1\ r.2$
			"add $r.1$,$r.2$"
Rule 2	$r.1$	$\rightarrow$	$+\ k.1\ r.2$
			"add $r.1$,$r.2$,$k.1$"
Rule 3	$r.1$	$\rightarrow$	$+\ r.2\ k.1$
			"add $r.1$,$r.2$,$k.1$"
Rule 4	λ	$\rightarrow$	$:=\ r.1\ k.1$
			"store $r.1$,$k.1$"

The four grammar rules above describe a sample of patterns for matching a simple three-address IR language and its corresponding register-based target instructions. These rules would be processed by a modified LR parser generation tool to produce a state machine to be processed by an LR driver program. The symbol r denotes a general purpose register and k denotes a constant. The symbols following '.' represent semantic qualifiers. That is, they indicate whether two operands of the same type, such as register, are distinct. For example, if they have the same qualification then they refer to the same register. Other ways

of indicating semantic restrictions include the specification of commutative operations such as Rule 2 and Rule 3 above. In addition, specific number values for a constant k can be used to indicate a specific value that a target instruction requires. For example, an increment instruction would require the value 1 as one of the operands to the $+$ operator.

For a target instruction to be generated, the left-hand side of a rule matches an input IR instruction and is reduced to the right-hand side. This reduce operation emits the target instruction (defined on the very right of each rule) and returns the result location (i.e., register) on the parser stack. Thus, information regarding a particular code generation process flows only up rather than down. This is not surprising as LR is context-free. This means that the choice of applicable grammar rules is based entirely on the local syntactic context rather than on its surrounding environment. Ultimately, this might lead to suboptimal target code. However, applying subsequent peephole optimization passes may significantly improve this.

The first observation we make about the pattern specification above is that the IR patterns are described using the same operand types as the target assembly code they match. In fact, a restriction of this particular approach is that it is required that the IR vocabulary be a subset of the target machine vocabulary and that the operators of both the source and target have the same computational effect [Glanville, 1977]. In this sense, the chosen IR language must be nearly identical to the target instruction set. Naturally the source and target must have operators that are similar in meaning for code generation to be possible, however, requiring the operands (memories) to be equivalent is overly restrictive. This impedes the general applicability of the approach and restricts its use in two wats: the grammars tightly couple the IR and target architecture, and the grammar descriptions are not very reusable..

In addition, the prefix form of the IR instruction effects is biased towards the left operand of a binary operator. This can lead to suboptimal target sequences as the right operand is considered only after a successful match of the operator and left operand. GIST resolves these issues by making the instruction *semantics* the driving criteria rather than its

syntax, allowing memory equivalences to be defined from the source IR to the target, and by considering the properties of an IR tree in its entirety rather than a left-to-right depth first search, as we discuss in Section 6.5.2.

2.3.2 Attribute Grammars

Recognizing the limitations imposed by the Graham and Glanville approach, methods were introduced to increase the amount of semantic information available during LR parsing. Here we discuss the use of attribute grammars exploited by Ganapathi and Fischer [1982, 1985]. In this work, attribute grammars are used as a means for specifying IR patterns and associated target instructions they implement. An attribute grammar is simply a grammar where each non-terminal can be associated with particular labelled values. These attributes can either be *synthetic* or *inherited*. Synthetic attributes are used to pass semantic information up a parse tree whereas an inherited attribute is used to pass information down. As an input is parsed, attributes are evaluated and their results are used to annotate the resulting syntax tree and guide the parsing engine. Because more attention is given to semantic context this approach offers a significant improvement over previous parser generator techniques.

In this work, the instruction set architecture is represented by a set of attribute-grammar productions. These productions collectively form the input to a program named cG that generates a code generator for the described machine. *Addressing mode productions* specify the pattern of an IR addressing mode. These productions are used to generate the address locations where values reside. *Instruction selection productions* describe IR patterns that correspond to a target instruction sequence to be emitted upon a match. Each production is represented by a left-hand and right-hand side. The left-hand side is a single non-terminal along with its set of synthetic attributes. The right-hand side consists of terminals, non-terminals, disambiguating predicates (each with their respective attributes), and actions that correspond to target instructions to be emitted. The disambiguating pred-

icates are used to allow patterns to be matched based on the values of the attributes (i.e., semantic context). The following example shows how we might specify these production rules,

Rule 1	Address↑a	→	V↑b ADDR(BASE(↓b), OFFSET(↓b), ↑a)
Rule 2	Reg↑r	→	Address↑a GETREG(↑r)
			{ emit('lw ',↑r,',',↑a) }
Rule 3	Reg↑r	→	R↑r
Rule 4	Reg↑a	→	'+' Reg↑b Reg↑c
			{ emit('add ',↑a,',',↑b,',', ↑c) }

Each of the above production rules has a non-terminal on the left-hand side. This non-terminal is decorated with an attribute to be returned on the parser stack (e.g., ↑r). The right-hand side indicates the IR pattern to be recognized. In our example, the terminals V and R represent *variable* (values in memory) and *register* (values in registers) respectively. Other terminals that are recognized by their textual representation are enclosed in single quotes (i.e., '+'). The emit function takes a variable number of arguments to be emitted into some output stream such as a text file when the rule is matched. Functions such as ADDR and GETREG are implemented externally as operations on IR trees that are executed if a rule is matched. The results are then assigned to their attribute parameters.

The cG program generates from the attribute-grammar description a set of transition tables that are used by a driver program. The driver operates similarly to a typical LR parser: it reads input and uses the transition table to determine valid IR instructions. Conflicts during parsing are resolved using the disambiguating predicates that examine the parser stack and control the parsing. As parsing proceeds, shift operations copy attribute values onto the parser stack and reduce operations process production actions and synthetic attributes are returned to the left-hand side non-terminal symbol.

The production rules we defined above would recognize our three-address IR instruction `a = + b c` (in linearized prefix form). In this case we will assume that the variable

`a` has already been allocated a register and `b` and `c` reside in memory locations that are at offsets from the current frame. In a typical bottom-up LR parse first `a`, `b`, and `c` would be matched by Rules 3, 2, and 2 respectively. Because `a` already resides in a register Rule 3 is matched and the corresponding register is returned. For `b` and `c` we must move their values from memory to a register. Because their values reside in memory Rule 3 is not applicable; however, Rule 2 tries to recognize the input using Rule 1. Rule 1 matches as they are both variables in memory. This rule then calls the function ADDR to determine the address of the non-terminal V attribute *b*. This requires the base (BASE) address of *b* and its offset (OFFSET). The computed result is then stored in attribute *a* and returned as the overall result of Rule 1. At this point, we continue with Rule 2 by allocating a register (GETREG) and storing this register in attribute *r*. Lastly, Rule 2 emits the PowerPC assembly instruction to load a value from memory into the register *r*. *r* is then returned as the result of Rule 2. Finally, the '+' and '=' are recognized and Rule 4 can emit an add instruction to add the two allocated registers for `b` and `c` and store the result in the register allocated for `a`.

Although this approach is an interesting application of context sensitive parsing theory to code generation, its general utility is questionable. In particular, disambiguation predicates are primarily a product of the semantics of the IR in the context of a particular target instruction. To that end, much of the effort in determining whether one production is acceptable over another is implemented in external functions. These functions would need to be implemented directly in external code similarly to the manual methods mentioned earlier, thus reducing the overall effectiveness of the approach. This is not surprising as the technique is inherently syntax driven, whereas the requirements of code generation are based almost entirely on the source and target semantics. As such, GIST uses semantics as its main driving factor in determining which target instructions are capable of implementing each IR instruction.

2.3.3 Bottom Up Rewrite System (BURS) Techniques

Although the Graham-Glanville LR parser style code generator is a significant breakthrough in the realm of automatic code generator generators, their approach is not a complete solution [Ganapathi et al., 1982]. For instance, the IR is very low level and is tightly bound to the target instructions (especially addressing modes). For example, an assumption is made that the mapping between IR operators and target machine operators is one to one. This makes it difficult to port the compiler and its IR from one machine to another: changes must be made to the IR. Because the code generator is inherently syntax driven and the notation is in prefix form it is difficult to generate better code based on the local context of the operator and first operand.

To alleviate these problems, methods utilizing tree-rewriting techniques typically based on BURS theory were developed [Pelegrí-Llopart and Graham, 1988, Emmelmann et al., 1989, Aho et al., 1989, Hanson and Fraser, 1995, Fraser et al., 1992, Proebsting, 1995, 2002]. A set of IR tree patterns is specified in a description file. This description file is then parsed and analyzed to produce a code generator that implements tree pattern matching on IR trees from the front-end of the compiler and produce instructions for the target.

Using tree-pattern matching instead of parsing eliminates the left-to-right bias problem found in previous approaches. In addition, dynamic programming algorithms [Aho et al., 1989] for determining optimal target trees have been integrated into these techniques for choosing optimal pattern rules for any point in the source tree based on provided cost factors. In addition, pattern rules can be specified in any order and the number of patterns that must be specified is significantly less.

Tree patterns are specified using *terminals* and *non-terminals*. The terminals represent operators in the source IR language and non-terminals denote a pattern tree. Tree patterns are associated with a cost (i.e., how favorable this match is) and semantic actions that, for example, allocate registers and generate target instructions. A BURS style code generator typically operates in two passes. The first pass works bottom up and finds patterns that

cover the tree with minimum cost. The second pass executes the semantic actions that are associated with the minimum-cost patterns. Consider the following list of pattern rules:

1. *var* ← Var
3. *reg* ← Indir(*disp*) (1)
4. *const* ← Const
5. *rc* ← const
6. *rc* ← reg
7. *reg* ← Add(*reg*, *rc*) (1)
8. *reg* ← Add(Indir(*var*), *rc*) (3)
9. *stmt* ← Asn(*disp*, *reg*) (1)
10. *disp* ← Addr(*var*)

Each rule specifies a possible rewrite that can be performed given a source IR tree. To the right of ← is an IR tree pattern. The symbols that have their first letter capitalized are terminals (operators of the IR) and the lowercase symbols correspond to variables or non-terminals that can represent any subtree specified by those rules were the non-terminal resides on the left-hand side. Costs for matching a particular rule are indicated by a (*n*) at the end of the rule. A rule without a cost is assumed to be 0. We left out the specification of pattern actions as they do not help in understanding the BURS approach. Figure 2.3 shows how a code generator generated from a set of BURS rules would operate on an IR tree for the code `a = b + 1`. The first pass would label the tree in a bottom-up fashion with the matching rules. In addition, it would compute a cost based on the sum of the rule it matched and any *chain rules*. A chain rule is simply those rules that can be reached through the use of a non-terminal on the right-hand side. For example, the rule Indir(*disp*), uses the non-terminal *disp* as part of its definition and thus reaches any of the *disp* rules.

In Figure 2.3, the BURS algorithm starts by matching and labeling the nodes for **Var** and **Const**. Both `a` and `b` are **Var** nodes and thus match rule 1 with cost 0. The integer `1` is matched by Rule 4 and also has cost 0. Through transitivity, it also matches chain

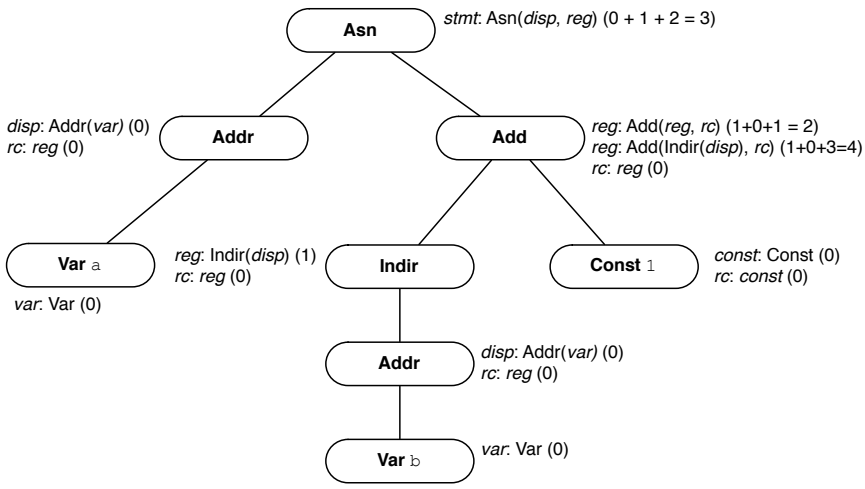


Figure 2.3: Matching an IR tree with BURS rules

Rule 5 with cost 0. With the leaves of the tree matched, the BURS algorithm can now proceed up the tree further matching nodes. The **Addr** node above **Var** `b` is matched by Rule 10 with cost 0 and Rule 6 through chaining. The **Indir** operator matches Rule 3 with cost 1 and Rule 6 through chaining. The **Add** operator is matched by Rule 7 with cost 1, Rule 8 with cost 2, and Rule 6 with cost 0 through chaining. On the left-hand side of the assignment we match the **Addr** operator with Rule 10 with cost 0 and Rule 6 with cost 0 by chaining. Lastly, the **Asn** operator is matched by Rule 9 with cost 1. Once the tree is labeled, the BURS algorithm will choose those matches that produce the least overall cost. The total cost for a particular match is shown between parenthesis at the end of the rule. The total cost is the sum of the cost of the matched rule and the costs of any child nodes. For example, The **Add** node matches Rule 7 with a total cost of 1+0+1=2. That is, the cost of 1 for its left subtree (**Indir**), the cost of 0 for its right subtree (**Const**), and the cost of 1 for itself, respectively.

A number of compiler frameworks use BURS technology for generating instruction selectors in an automated fashion. For example, the tool *JBurg* is used for generating an instruction selector for the Java bytecode optimizing compiler in JikesRVM [Alpern et al., 2000a]. Similiarly, the instruction selector patterns for the LCC C compiler [Hanson and

Fraser, 1995] is described by BURS rules. It allows an implementor of a code generator to specify patterns that match IR code fragments. These patterns are specified along with action routines that emit target assembly or binary code and perform register allocation as pattern rules are matched. In addition, the technique chooses the optimal patterns based on cost factors using a dynamic programming algorithm. Unfortunately, the implementor must specify the action routines. That is, we still require a significant amount of knowledge of the compiler *and* target machine to map IR patterns to target instructions and memories. Our approach alleviates this process through heuristic search in the space of target instruction semantics to find these pattern mappings. To leverage the efficiency of BURS, however, it is possible (but not required) that GIST output the mapping it discovers as BURS rules. Indeed, this is how we will evaluate our approach for LCC: replace the hand-written BURS rules of LCC with those GIST generated automatically.

2.4 Automatic Generation by Pattern Discovery

In the previous sections we looked at techniques for constructing code generators in an automated fashion by specifying IR patterns and corresponding target instructions. In this section, we look at past approaches for automatic discovery of IR patterns and the target instructions that can be generated from them. In particular, given a sequence of IR instructions, some form of heuristic search is employed to find a sequence of target instructions that perform the same function. A successful search yields a set of *instruction pattern mappings*, pattern rules that map IR instructions to target instructions. In the first part, we discuss methods for generating these pattern mappings when the source and target machine are the same. The second part looks at prior work for generating pattern mappings when they differ.

2.5 Same Machine Pattern Mappings

A special kind of code generator is one that attempts to generate more efficient instructions for some machine M given a sequence of M instructions. To this end, the IR instructions are the same as the M instructions. This kind of code generator, often referred to as a *Superoptimizer*, is often used for building peephole optimizers [McKeeman, 1965]. A peephole optimizer scans a sequence of instructions (usually within a basic block) with a fixed window, say 2 instructions wide. The optimizer keeps a dictionary of instruction pattern sequences (in this case, 2 instructions wide) that map to a more efficient sequence of instructions (often just one) on the same machine. Although it is easy enough to specify these patterns manually, it is advantageous to have them automatically generated by a superoptimzer, especially if some of the patterns are not obvious.

2.5.1 Massalin

For example, the first superoptimizer program written by Massalin [1987] was able to discover instruction pattern mappings for Motorola's 68020 instruction set. A striking result of this work was the obscurity of the patterns and how they were found. For example, the generated mappings were able to take a sequence of 9 instructions for a function that determined if its argument was positive, negative, or zero and generate code for the same machine that accomplished the same effect in only 4 instructions. As it turns out, the superoptimizer found that it could exploit two's complement arithmetic to eliminate all the branches in this code. In particular, it was able to utilize the setting of the carry flag based on the semantics of particular instructions to return the correct result.

The basic approach taken here is to search for target instructions (in this case, the same machine) that accomplish the same effect as the original instruction. This is accomplished by an exhaustive search in the space of instructions. In particular, all combinations of the instructions are enumerated, first of length 1, then of length 2, and so on. Each of these generated programs are then tested against the source instructions. This probabilistic test

consists of running the source and target instructions on a set of test inputs on the underlying machine. If the outputs of the generated instructions are the same as those of the original instructions, in practice we can assume with high probability that they are semantically the same. A more formal boolean test was also investigated, but it took too long to compute the boolean expressions for every program generated. As such, the work was viewed as being an aid to assembly programmers rather than as a general purpose peephole optimizer generator.

2.5.2 GNU Superoptimizer

This work was followed up some time later with the GNU Superoptimizer (GSO) [Granlund and Kenner, 1992] which uses superoptimizer techniques in an attempt to eliminate branches in the GNU C Compiler (GCC). The GSO approach improved on Massalin's work in several dimensions: portability, configurability, and search strategy. The original superoptimizer was limited in that it was written in Motorola 68020 assembly language and could target only the Motorola 68020. GSO is able to generate code sequences for a variety of architectures. This portability is accomplished by defining a set of generic operations to cover all the instruction sets targeted by GSO. These generic operations are then interpreted by a C program.

Configurability is achieved by having a minimal number of additions to the GSO infrastructure to add a new architecture. First, generic operations for any instructions present on the new machine but on no previous machine must be added and the interpreter must be augmented with their interpretation. The search procedure must be modified to reflect which new operations exist for the new machine. Code must be written to output the operations in assembly.

The GSO search strategy uses more heuristics to choose potential replacement instructions in order to reduce the search space. For example, instructions are chosen that have some kind of relationship between them. That is, if we are generating all instruction se-

quences of length 2, do not generate those sequences that do not make sense. An instance of a 2 instruction sequence that does not make sense is one where the second instruction uses the carry flag in its computation, but the first instruction does not affect the carry flag. In this case, such a useless sequence would not be considered. Other heuristics include choosing instructions only if its operands have been inputs to or outputs from a previous instruction.

The GSO is an improvement over Massalin's work, but still lacks the level of certainty that is required for generating code. It must be the case that we can prove without a doubt that the generated code is identical in meaning to the original source. This is true whether we are generating code for machines that are different or for those that are the same. The choice to interpret virtual operators is a good one in that the technique is portable to the number of target machines targeted by GSO. In this dissertation we present the CISL machine description language that provides a universal set of operators for describing instructions. One can imagine using CISL and its well defined semantics to implement a semantic interpreter to create a GSO-like superoptimizer.

Reducing the search space is an important consideration when attempting to discover equivalent instructions. It is easy for the search space to explode given a real instruction set and a large set of test values over which to interpret instruction semantics. Massalin was unable to use this information as no formal language existed, and the approach in GSO is ad hoc. The semantics of the CISL language could be used to reduce the search space further in a GSO approach—we leave this investigation to future work.

2.5.3 Denali

Recognizing that a formal semantics is an important factor in generating code generators and that the correctness of the generated code is important, Joshi et al. [2002, 2006] investigated a goal-directed superoptimizer called *Denali*. They used automated theorem proving techniques to search for instruction sequences for performance critical code fragments. In

contrast to Massalin's work, it was recognized that passing tests is not the same as being correct. To this end, the idea is to use a *refutation-based* theorem prover to prove a conjecture C by establishing the unsatisfiability of its negation $\neg C$. In particular, to generate optimal code for a program fragment P, they express a conjecture in formal logic in the following form:

> No program of the target architecture computes P in 8 cycles or less.

This conjecture is then submitted to an automated prover. If the proof succeeds, then 8 cycles are not enough, and a new conjecture is proposed with 16 cycles. If the proof fails, then the failed proof shows an (at most) 8-cycle program that computes P. They apply this procedure iteratively with a new conjecture stating that a 4-cycle program does not exist for the generated program from the previous proof. Eventually, the optimal program is found, for some k, a k-cycle program that computes P.

Denali programs are described using a C-like language and is limited to small code fragments such as critical inner loops and short procedures. This language is then translated into a sequence of so called *guarded multi-assignment*s (GMA). The anatomy of a GMA is simply: *guard* $\rightarrow$ (*targets*) := (*newvalues*), where targets is a list of *lvalues*, newvalues is a list of expressions, and the guard is some boolean expression. Thus, the targets get the values of the newvalues if the guard is true.

The above is the Denali ideal; the actual approach differs in its implementation. In particular, it uses *matching* to determine all equivalent code fragments given an input code fragment and a set of *axioms*. These axioms define mathematical equivalences such as $\forall k, n : k * 2^n = k << n$, as well as the operations relevant to a particular target architecture,[2] and are used to instantiate all possible ways of computing the input code fragment (for a specific target). Given this set of alternative programs and an architectural description, the conjecture mentioned above is formulated in propositional calculus. The architectural de-

[2] Although these axioms are basically a mapping from the Denali IR to target instructions, the goal here is to find patterns for the optimal target instructions.

scription includes tables specifying which functional units can execute which instructions and a table of latencies of the various ALU operations.

Although the overall goal of our work is different to that of Denali, the adherence to a precise language and importance of generating correct code is not. In addition, their use of matching and rewriting as well as boolean satisfiability is particularly relevant. Denali is using these theorem proving techniques to determine optimal code sequences whereas our application of these methods is in generating canonical representations of source and target semantics.

2.5.4 Bansal and Aiken

Another approach to superoptimization is the recent work of Bansal and Aiken [2006]. They introduce techniques that reduce the overall search space of *candidate* instruction sequences (those that might be equivalent to the original sequence in question) and an efficient test for equivalence between the original and the candidate sequence. More specifically, a *harvester* extracts instruction sequences from a set of training applications and an *enumerator* exhaustively enumerates all possible candidate sequences up to a certain length, checking if each might be an optimal replacement.

The harvester extracts sequences of instructions from training applications that meet the following constraint: an instruction sequence I must have a single entry point—no instruction in I (except the first) should be a jump target of any instruction outside of I. The sequence includes all instructions thereafter that are not jump targets. It is important to note that a harvested sequence can have multiple exits since jump instructions are allowed in the sequence. The number of live registers at the end of the sequence is also recorded and used to determine equivalence by the enumerator.

To ensure that duplicate sequences do not exist, it is noted that many sequences are just transformations of each other under renaming of registers and immediate operands. As such, each sequence undergoes a process called *canonicalization*. In particular, a sequence

is canonical if its registers and constants (not literals) can be renamed in order of appearance with a given instruction sequence. To accomplish this, a set of symbolic registers (starting with *r0*) and constants (starting with *c0*) are used to rewrite the original sequence. In addition, the observation is made that an optimization that applies to a specific sequence is also valid for its canonical form.

Instruction sequences are enumerated from a subset of all instructions by the enumerator. In particular, at most one branch instruction is allowed. That branch instruction, targets a canonical branch target representing the exit point outside of the sequence in question. In addition, the number of distinct registers and constants is restricted. For instance, for instructions that apply only to a subset of the registers for a particular machine, only those registers are considered. This, in turn, restricts the number of indirect memory accesses to those that use the number of allowable registers.

Because the search space of the enumerated sequences is exponential in the length of each sequence, two primary approaches are used to reduce that space: only canonical sequences are used and instructions that are functionally equivalent to other cheaper instructions are eliminated. Following the reduction process, instruction sequences are stored in a table along with information about the registers and constants that are used. In addition, a *fingerprinting* technique is used to allow for efficient lookup.

This table is then used to locate instruction sequences that may implement the sequence discovered by the harvester. If a match is found based on register and constant use, two techniques are used to determine if they are equivalent semantically. The first approach is to execute on the underlying machine the harvested sequence and the enumerator sequence on a set of test vectors. If the output of each is identical, the two sequences might be equivalent. In that case, each is translated into a boolean formula that expresses the equivalence relation as a satisfiability constraint. This formula is then provided as input to a SAT solver to determine equivalence.

The boolean formula is represented by a finite set of registers and a model of the memory and stack. Registers are translated into a vector of boolean variables and memory is modeled by a map from address expressions to data bits. Our approach differs in that we rely on a canonicalization (described in Section 5.1) process and a set of transformation axioms that define semantic equivalences. Although these axioms tend to be simple and in many circumstances can be proved correct by inspection it is possible to translate an axiom into a boolean formula and provide that as input to a SAT solver to determine its validity. To do this, we can encode references to memories into a boolean formula based on an initial unspecified state. Each operation is represented as a boolean circuit transforming an input machine state to an output machine state. Branch targets are handled by predicating the "execution" of instructions on the true and false paths (i.e., the branch condition or its negation). The original input state is shared between the two circuits representing the left- and right-hand side of the axiom. The left- and right-hand side of an axiom are equivalent iff the resulting memories in the final states are equivalent. Although we have preliminary results demonstrating the feasibility of translating our machine description language (CISL) into SAT formulae we leave this to future work.

2.6 Different Machine Pattern Mappings

In the previous section we reviewed prior work in generating instruction pattern mappings for code generators targeting the same machine as the source. In this section, we look at techniques for generating these pattern mappings when the source and target instruction set are different. In particular, these approaches pertain to generating code from a compiler IR whereas the previous techniques merely rewrite instruction sequences. In that respect, the following work more closely aligns with the goals of this dissertation.

2.6.1 XGEN

An early attempt at generating code generators automatically was the XGEN system presented by Fraser [1977] in his dissertation work. He argued that a code generator could be constructed in an automated fashion by capturing human skills and knowledge in rules that formed the basis of a *production system*. These rules are implemented using patterns over the IR of the compiler, called XL, and an ISP machine description formalism [Bell and Newell, 1971]. Thus, to generate a code generator requires a set of rules about code generation and a description of the target machine. Porting the compiler to a new architecture, then, requires a corresponding machine description and possibly additional rules to capture the obscurities of the new machine.

To generate a code generator, XGEN first performs some analysis of the ISP description it takes as input. In particular, registers are classified in various ways, such as, accumulators and index registers. Memories are analyzed to determine width and alignment with respect to the various datatypes allowed in the XL language. And instructions are categorized according to their type (e.g., branch, move). This information can then be accessed by the XGEN rules to infer instruction patterns with respect to the properties of the machine.

The code generator rules are evaluated by a production system to determine the sequence of target instructions that is to be generated given a sequence of XL instructions. A production system is composed of a set of rules and an associated state. A rule is associated with a set of assertions and an action. A rule is said to "fire" if its assertions are true based on the state of the system. A rule that fires executes its action, which can modify existing state and/or introduce new state that can be used by other rules.

For the XGEN system, rules encode the steps a human implementor must take in order to build a code generator by hand. The state represents those elements that are to be remembered, such as IR patterns that have already been identified and registers that have been allocated for a particular sequence. As rules are matched and fired, code generator patterns

are generated that map IR to target instructions. The following is the English version of a rule taken directly from Fraser:

> If compiling some operation with an operand that is not in an accumulator and if the operation otherwise matches some instruction then allocate an accumulator, load it with the operand, and change the original operation to reference it.

Thus, this rule allocates an accumulator and loads the operand of a particular operation with it *if* the operand is not already in an accumulator and it does not already match a target instruction. Of course, loading the accumulator with the operand matches a particular instruction based on the initial analysis of the machine description. It should be clear that a rule such as that mentioned above would pertain only to an accumulator based instruction set (in fact, one of the machines targeted by Fraser was the PDP10).

The application of production systems to code generator generation is an interesting one. Indeed, it follows the process the human implementor must take in order to build a code generator manually. That is, patterns on the compiler IR are recognized and translated into corresponding code sequences on the target. In addition, the ability to add rules incrementally to the system allows for extensibility and incremental development. That is, a simple code generator could be produced given a set of rules. These rules could then be extended to special-case particular patterns to produce better code sequences.

Although the overall approach taken by Fraser is unique, there are several drawbacks. In particular, the quality of the generated code is limited by the available rules. Of course, one could always extended the system with a new rule that captures a particular optimization pattern. Unfortunately, as many of the Superoptimizer approaches from the previous section mention, optimal code sequences are not entirely obvious and require a deep understanding of the underlying properties of the machine. As such, human skill and intuition might not weigh as heavily as heuristic search and the generate and test approach.

In addition, Fraser maintained that as more machines are targeted using his XGEN system, it decreased the number of rules required for each additional instruction set. Unfortunately, his argument was based on the fact that those machines were closely related. Thus, adding an entirely new kind of architecture still requires significant work in terms of rule creation. This, in itself, is problematic in the sense that not only must compiler writers have a mastery of the compiler IR and target instruction set, they must also be well versed in the area of production systems and XGEN rule creation. Perhaps this is not much of a problem if the diversity of machines is relatively low, but it might be more of an issue in an experimental setting.

A practical problem with the XGEN approach is that it relies heavily on the structure and interpretation of the XL compiler IR. Thus, to use the XGEN system requires the use of the XL intermediate representation. This is problematic as compiler implementations use a variety of IRs and thus restricts the approach to new compilers utilizing the XL language. GIST removes this restriction by permitting[3] the compiler IR to be specified by a machine description. As such, the approach is both compiler- and architecture-independent.

Another problem associated with the XGEN system is related to the correctness of generated code. There is no firm way to know if the instruction patterns generated are correct as no formal methods are applied. This is in contrast to other approaches that have used SAT solvers to determine the equivalence between instruction semantics. Although Fraser agues that the application of formal methods has the potential to be combinatorially explosive, recent methods have shown that with careful application they can be used in practice. It is important to mention, however, that a correct code generator is better than a code generator producing broken code. Furthermore, complexity of debugging incorrectly generated code is dramatically increased as the IR instruction patterns are intertwined with the firing of rules in the production system.

[3] Requiring, actually!

Lastly, the XGEN implementation is interpreted. That is, the ISP description and rules are evaluated at compile time rather than preprocessed to produce a table of patterns at compiler compile time. This leads to a prohibitively slow code generator and limits its application with respect to the wide use of JIT compilation techniques today. Although Fraser does mention that XGEN is not restricted to interpretation and could be adapted to a preprocessing strategy, it does not appear to have been followed up.

2.6.2 Cattell

Another main contribution in this area is the thesis work of Cattell [1978]. The goal of his work was to test the feasibility of taking a formal approach towards automatic generation of code generator patterns. In particular, a formal model of a machine is specified and a set of axioms are used to define the semantic equivalence of instructions between the IR used in the PQCC compiler [Wulf, 1980] and the instructions of the target machine. These axioms are used to rewrite a sequence of IR instructions into a corresponding sequence of target instructions. Thus, in addition to generating an IR pattern that maps to a sequence of equivalent target instructions, it generates a proof of that mapping in the form of valid rewrites.

The machine formalism used by Cattell is genealogically related to the ISP descriptions mentioned previously in Fraser's work. In particular, it allows for the definition of machine memories such as registers and main memory including their size and alignment constraints. Instruction semantics, referred to as *machine operations* (MOPs), are specified using a LISP-like notation and includes support for specifying *access modes* (addressing modes) as well as classes of addressing modes named *operand classes*. Instructions are additionally annotated with their binary encoding and assembly level syntax for purposes of actually generating target code.

Given a machine description of the target machine and a tree-based IR (TCOL) instruction, an attempt is made by heuristic search to generate code sequences for IR operators

that do not correspond to a single target instruction. A search procedure recursively visits the nodes of the IR tree examining the operators and operands of each subtree. If there is an operator mismatch, a set of semantic-preserving tree equivalence axioms are applied to the offending operator in an attempt to rewrite the subtree into something that may enable search to proceed further.

The axioms specify a wide range of semantic equivalences. In particular, they cover arithmetic and boolean laws (e.g., DeMorgan's laws), rules about memories, side effects, store access, and sequencing semantics. An axiom has the form: $l :: r$, where l and r are tree patterns. Tree patterns, also referred to as *parameterized trees*, are used extensively in Cattell's work. A *parameter* is a variable associated with a particular node in a tree pattern. When a tree pattern matches all or part of an IR tree the parameter variables represent the tree node that matched the pattern node and can be referred to later. Consider the following pattern tree (using the same notation as Cattell):

(← $1:Reg (+ $1:Reg $2:Opnd))

Here the symbols $1 and $2 are the parameter names. $1 is used to indicate that the source and destination registers can match any register specified by the operand class Reg. In addition, the use of the same parameter $1 in both the source and destination indicates an additional constraint, namely, that they must be the same register. Thus, an IR tree can be matched by the above pattern given the parameters and their constraints. The variables can then be used in later processing to refer to those subtrees that were matched. For example, the axiom pattern below specifies a tree transformation where the left-hand side is a pattern to match on a given IR tree and the right-hand side is the transformation that is applied with substitution of variables:

(AND $1 $2) :: (NOT (OR (NOT $1) (NOT $2)))

To reduce the search space, the axioms are categorized into three classes: *transformations*, *decompositions*, and *compensations*. The transformation axioms are those that are

concerned with arithmetic and boolean equivalence, decompositions are rules regarding control constructs and storage locations, and compensations are focused on side effects. Each category of axioms are applied at different points in the search process to achieve a different goal.

The most influential method used in the search process employs a well known artificial intelligence technique known as *means-ends analysis*. This technique is used to decide how to get from some starting state to a goal state by identifying the difference between the two and choosing some action that reduces that difference. In this case, the goal is to find target instructions that are semantically equivalent to the source IR. The starting point, of course, is the given IR tree. Because the actual target instruction sequence is unknown, a set of possible target instructions are chosen based on their semantic representation[4] and a set of heuristics, such as choosing target instructions whose primary operator is the same as the source IR. In addition, instructions are chosen whose operators are related (e.g., + and -). This is achieved by using the transformation axioms to rewrite the IR tree into an equivalent form that uses a different operator.

Each possible target instruction found based on operator matching of the IR and target semantic trees is ordered by source operator and then by source operands to facilitate choosing more likely matches first. For each of the feasible target instructions found, in the specified order, an attempt is made to transform the source tree to match the target tree. More specifically, for each target semantic tree, try to match the IR tree node by node. If a mismatch occurs, select the transformation axioms that apply to the IR tree and rewrite into new IR trees. These new IR trees are then used to apply further matching and transformations.

In addition to transformation axioms for handling differing operators, rules exist for transforming operands. Consider the case of the IR tree that is adding a value contained in

[4]Because the semantic descriptions of the target instructions are specified with the same semantic notation as the IR used in the PQCC compiler, semantic comparison is simpler. Unfortunately, this also limits the general applicability of the approach as it is tied to a particular compiler.

some memory location and a temporary/register. Now imagine the target is some RISC-like processor that does not support register-memory addition directly (i.e., the value in memory must first be loaded into a register and then addition is performed on the register contents). As is, there would be no matching target instruction and search could not proceed. To fix this problem, a pair of transformation axioms named *fetch/store decomposition* are provided. These specify transformations on trees to introduce movement of operand values. Thus, an operand that does not match can be moved into a location that does match. For the case above, this would involve moving a value from memory into a register location and using that register as the new operand. If the operand mismatch was located on the left-hand side of an assignment (i.e., an addition that stores the result in memory) this could equally be transformed into a data movement instruction that occurs after the operation (i.e., store a register value into memory).

Naturally, it is possible that no target instruction matches the source IR tree's operator and operand(s). To alleviate this problem, decomposition axioms are used to transform a given IR tree into more primitive control structures. For example, an *if* conditional may be transformed into a *label* and *goto* construct. The hope here is that if complicated conditional tests are used in the IR trees that do not match any target instruction semantics, decomposing these into more primitive constructs may enable search to find a corresponding target instruction for each of the smaller pieces. Because most machines have several branching instructions this is a sensible approach.

In order to provide efficient access to the list of target instructions during code generator generation Cattell uses compensation axioms. These axioms are not specified directly as the previous two cases are, but are really a set of heuristics that are used to preprocess the set of target instruction semantics. For example, if the semantics of a target instruction involves more than one effect, special analysis is performed to determine if any one of those effects can be utilized separately. For example, an instruction may set condition code registers in addition to its primary effect. This kind of instruction can certainly be

exploited without using the condition codes; however, care must be taken to ensure that any instructions generated afterward are aware that the condition codes have been modified. A compensation axiom thus introduces an *allocation* of those condition code registers to inform the search procedure of this case.

Using these techniques, Cattell was able to find target instruction patterns for IR trees that had no corresponding operation on the target. For example, the system was presented with an IR tree for a memory-memory AND operation. The task was to locate a sequence of instructions on the PDP-11 that performed the same computation. The PDP-11, however, does not have an AND operation and thus requires axioms to transform the input tree. The search process discovers that this operation could be performed by using the COM (complement) instruction followed by BIC (bit clear). That is, complement one of the arguments and then subsequently use the BIC instruction to AND the complement of the complemented argument with the other argument. This uses different operations but is semantically equivalent.

Cattell was able to generate correct code generators automatically for a variety of architectures including the IBM 360, PDP-10, PDP-11, Intel 8080, Motorola 6800, and the PDP-8. It does not appear, however, that anyone followed up on this work to determine its applicability more generally. In particular, the description language used by Cattell was based on the tree language TCOL used by the PQCC compiler. To this end, it made the descriptions highly compatible with a specific compiler. Although useful for exploring the viability of the approach, its general use is questionable.

Although the fixed compiler issue may have prevented wide-spread adoption of the technique, other problems exist. For example, the compensation axioms and heuristics were integrated into the search algorithm. This makes it difficult to alter them without changing the algorithm. As such, making additions to the compensation axioms or heuristics to accommodate new architectures could reduce the reliability of the system.

In addition, the system assumes there exists a *move* instruction in the target instruction set between all possible memory locations to be utilized by the store/fetch decomposition axioms. If not, it is possible for the search algorithm to block the generation of target patterns given a valid IR tree. In the best case, this may involve generating unnecessary moves simply to get an operand into the proper location.

Another issue involves the use of memory state during the search procedure. In particular, it fails to recognize equivalent locations such as a value that exists both in memory and a register. This results in emission of possibly redundant or useless operations, such as reloading a value from memory into a register. This is primarily caused by the local context in which tree matching occurs. That is, a search on a subtree of the IR tree loses information with respect to what was previously generated by the parent tree. Of course, a separate peephole optimization pass might improve this situation. GIST eliminates this problem by transforming the memory mapping to reflect these situations. As such, it provides a global context to enable better target candidate selection.

Lastly, there is potential for this approach to be combinatorially explosive as axioms are applied arbitrarily across all nodes over all combinations to an IR tree. Cattell controlled this behavior by limiting the depth of the search. While this worked, it can potentially miss better sequences.

2.6.3 Dias

A more recent proposal for generating the back-end of a compiler automatically is the dissertation of Dias [2008]. He describes an approach for generating the back-end of a Davidson/Fraser style compiler [Davidson and Fraser, 1984a] (DFC) using a *declarative machine description*. The DFC approach is a portable optimizing compiler that relies on a machine-independent intermediate representation called *register transfer lists* (RTL). Unlike a typical IR, RTL can be used to represent expression trees rather than single operations as is the case in a typical IR such as tuples or 3-address code. Also unique to this approach

is that target instructions are chosen early in the compilation process and are represented by RTLs. These RTLs are then subjected to a machine independent optimizer to improve target instructions. Throughout the optimization process the compiler ensures that each RTL can be represented by a single instruction of the target machine—this is called the *machine invariant*.

Dias uses RTL as the basis for describing instructions on the target and the IR form that is generated from the C−− compiler back-end [Jones et al., 1999]. The first phase of the DFC expands compiler RTL (representing the compiler IR) into instruction RTLs for a specific machine. The "expansion" process is performed by the *expander* which is factored into a machine-independent procedure that identifies sub-graphs within an RTL control-flow graph called *expansion tiles*. The traditional DFC approach is that the back-end must provide a machine-dependent implementation of each expansion tile using only RTL from the machine instructions. It is important to note, however, that the set of expansion tiles remains the same across all target machines. Dias shows how the implementation of the tiles can be discovered using search and a description of the target instructions in λ-RTL (a language layer above RTL). The basic approach is to match target RTLs to the machine-independent tile set. If a match cannot be found for a tile, target RTLs are intelligently composed to form larger RTL expression trees that can be used for tile recognition.

After the RTL has been processed by the expander, the machine-specific RTL is then used as input to a DFC phase called the *recognizer* whose job is to decide if an RTL can be implemented by a single instruction on the target. To do this, the RTLs from the source must have compiler objects (e.g., pseudoregisters, compile-time constants) translated into locations and literals that are acceptable on the target machine; Dias calls this "bridging the semantic gap". After translation, a pattern matching process is used to match compiler RTL to target RTL patterns. Unlike GIST, the matching technique is simple and does not recognize equivalent expressions under the laws of associativity and commutativity. A canonicalization process is applied to improve matching; however, it is simple and applies

only to reducing compile-time expressions into constants. The simplicity restriction during the recognizer phase is important because the approach occurs at compile time not at compiler compile time as is the case for GIST. The custom recognizer is generated for each target using a BURG-style matcher. The recognizer generates assembly language as output.

Although this approach is similiar in spirit to ours (strives to be compiler- and machine-independent) it differs in that it is applicable only to Davidson/Fraser style compiler frameworks and that it works directly from the RTL that is generated by the front-end (rather than a description of the IR in λ-RTL). In our approach we do not depend on an early translation of IR into machine-specific RTL code and hope to generate instruction selectors that can be adapted to a variety of requirements and frameworks. In addition, we describe both the target instructions and compiler IR using CISL. In contrast to GIST, the Dias approach works by matching target RTL to a predetermined set of expansion tiles. In essence, this works from target to IR where the GIST search procedure works from IR to target. Furthermore, our search process attempts to match all subtrees of an IR instruction to a sequence of target subtrees—each IR subtree corresponds to a target instruction. It is not unreasonable to imagine using CISL trees to compose larger trees to match larger IR subtrees more quickly, thus improving search time and eliminating the need for the machine-independent expansion tiles.

CHAPTER 3

DESCRIBING MACHINE INSTRUCTIONS

3.1 Introduction

Automatically generating instruction selectors and other machine-oriented software requires a clear definition of the machine. The definition must be rigorous enough to be manipulated by a program, yet flexible enough to allow one to define a wide variety of machines, including not only traditional processors such as the IA-32 [Intel Corporation, 2003] or PowerPC [May et al., 1994] but also ISA extensions or wholly novel architectures.

With enough flexibility, the same language can be also be used to describe the source IR (or VM). For this reason, we introduce the CISL language because it provides the combination of rigor and flexibility needed to represent all of the GIST inputs and outputs as shown in Figure 1.6. This is a novel feature of GIST: all previous systems use a *single IR*, of their own definition, and thus can *not* be used directly with multiple existing compiler IRs.

Using CISL to describe both the IR and target facilitates the building of instruction selector patterns: we express the source and target in the same language, whose semantics GIST "understands," allowing it to be rigorous about the semantic equivalence of mapped source instructions and target instruction sequences (By "mapped source instructions" we mean source instructions whose storage resources have been mapped to target resources using the store mapping given by the GIST user as described in Section 3.2.2.)

Many others have devised machine description formalisms for specific applications [Davidson and Fraser, 1984b, Bell and Newell, 1971, Joshi et al., 2002, Fauth et al., 1995, Ramsey

et al., Kessler, 1986, Farfeleder et al., 2006, Hoover and Zadeck, 1996, Ramsey and Davidson, 1998, 1997, Ramsey and Fernandez, 1997], most of which grew out of the needs of a particular problem domain. Although our formalism also supports automatic generation of other systems-related tools, such as functional simulators, assemblers, and symbolic debuggers, [Richards et al., 2007], we focus here on aspects related to instruction selector pattern generation.

Similar to other formalisms, our view of a machine consists of a set of locations called the *machine store*, a list of *instructions* that manipulate the store, and an *execution loop* that executes a particular instruction based on the current contents of the store. For instruction selection, the most important aspect is the definition of an *instruction*, which represents a low-level machine-oriented operation capable of being executed by a processor. CISL thus describes operations of modern processors, compiler IRs, and virtual machines, but not higher-level semantics embodied by an abstract syntax tree (AST) that represent high-level programming language constructs (e.g., object field access) that are generated by the compiler front-end (these must be decomposed into machine-like operations referencing actual machine locations). Similarly, one needs to resolve abstract operations dependent on separately defined semantics such as calling conventions (e.g., `invokestatic` on the Java Virtual Machine (JVM) [Lindholm and Yellin, 1999]) or specific runtime calls into a CISL specifiable form using other formalisms [Lindig and Ramsey, 2004] or an adapter (elaborated on in Chapter 7).

3.2 Overview

We define an instruction in CISL using three important machine abstractions: *store*, *encoding*, and *semantics*. The store specifies the set of locations available on a particular machine and how an instruction can access them; these are collectively referred to as the "machine state". The encoding defines the binary representation of an instruction including size (in bits), fields (e.g., opcode), and fixed values. The encoding information is not required by

GIST, except for interfacing to details in an IR, and is largely symbolic in the semantics. The semantics describe the computational effect of an instruction on the machine state. It is important to note that CISL does not describe assembly syntax or other formats that relate to a particular tool or environment; its sole purpose is to describe *only* those aspects that are relevant to the semantics of an instruction. We expect other related description languages, such as those depicted in Figure 1.3 and adapters (Chapter 7), to interface with CISL to generate target applications. In the rest of this section we highlight the basic structure of a CISL semantic description; in Sections 3.2.1 to 3.3.3 we provide a thorough discussion of the CISL language.

An instruction in CISL is a sequence of guarded effects. A guard represents a *condition* on the state of the machine and an effect is a side-effecting operation that is "executed" if the guard is satisfied. We can expression this notion as a conditional assignment statement similar to that found in most programming languages:

```
if G1 then E1;
if G2 then E2;
if G3 then E3;
...
if Gn then En;
```

where G_k and E_k are the guard and effect for the conditional statement k. For the purposes of this work, we represent guarded effects as *semantic pattern trees* (similar to the AST representation used by the front-end of a compiler). This representation is more easily used by GIST during heuristic search to match IR subtrees to target instruction trees. The details of the matching process are elaborated in Section 6.5.2. Consider the following snippet of CISL semantic code of the JVM **`if_cmplt`** bytecode:

Listing 3.1: CISL semantics for the if_cmplt bytecode

```
if (S[sp] <= S[sp-1]) {
  PC = target;
}
sp = sp - 2;
```

This bytecode pops two operand values from the stack and branches if the comparison ($\leq$) succeeds. The first guarded effect expresses the comparison operation between the top two values on the stack; setting the **PC** (program counter) to the target address if the condition is *true*. The second guarded effect updates the stack pointer (**sp**) to "pop" the operands off the stack. Note that in CISL one is not required to specify a guard if an effect always executes, GIST inserts the **true** condition automatically to canonicalize the semantic representation to match target instructions more effectively as discussed in Section 5.1. We express the semantics in tree form using a LISP-like s-expression:

Listing 3.2: Pattern tree representation of the if_cmplt bytecode

```
(seq
  (if (le (lget (mem S) (mem sp))
          (lget (mem S) (sub (mem sp)
                             (iconst 1))))
      (lset (lget (mem PC)) (param target)))
  (if true
      (lset (lget (mem sp)) (sub (lget (mem sp))
                                 (iconst 2)))))
```

The above pattern tree representation provides semantic information that is not visible in the CISL code in Listing 3.1. For example, it is now clear **S**, **sp**, and **PC** refer to CISL memory definitions. In addition, instruction parameters, also known as fields, are explicit as shown by the branch destination location (**target**). In addition to operations, memory references, fields, and literals, the tree form records type information for each expression which is important for determining equivalent semantics and constraints during matching (we elide the type information in the above example for the sake of readability). We discuss cisl types in Section 3.2.1 and describe their use during search in Chapter 6.

CISL provides a wide range of operators that are capable of describing the instruction semantics of a wide variety of architectures. This includes operators for standard arithmetic such as addition, subtraction, and multiplication as well as logical and bitwise operations. In addition, CISL is equipped with several bit manipulation operators supporting sign-extension, bit re-sizing, and population count. We have found the current set of op-

erations sufficient for describing a variety of machines for compiler IRs, VM instructions, and real architectures. We describe the operator set in Section 3.3.2.3.

Furthermore, CISL uses modern programming language techniques to reduce the effort required to describe complicated semantics. Consider the instruction set for the Intel x86. The *ADD* instruction can address a value in a variety of ways leading to approximately fifty unique definitions of the semantics. To describe each variation individually would be time consuming and lead to descriptions that are unwieldy. For this reason, CISL borrows concepts from object-oriented programming languages and allows semantics to be described using *methods*, *classes*, and *mixins*.

Just as we use methods to decompose an algorithm into smaller pieces, CISL uses methods to decompose a semantic description. We specify Related methods in a class to form part of an instruction semantic definition. A class can then inherit from another class to form a hierarchy of semantics that can be easily re-used and extended (e.g., x86). CISL supports single inheritance because of the problems that can arise when multiple classes can be extended.

Class inheritance is excellent at decomposing semantics along encoding formats of an instruction set. However, it is not convenient for expressing cross-cutting semantics such as addressing modes. An addressing mode defines how values are accessed on a machine (e.g., immediate, register, memory). CISC architectures (e.g., x86) may have a large number of addressing modes that can be used in a variety of ways across the entire instruction set. As mentioned previously, the x86 architecture has approximately fifty different versions of the *ADD* instruction, all performing the same operation: they differ in the addressing modes they use. To alleviate the burden of defining slight variants of the same instruction multiple times, CISL supports *mixins*. A mixin contains methods that define cross-cutting semantics. Multiple mixins can then be "used" by a class, at different points in the class hierarchy, to reference mixin methods.

The application of modern programming language techniques in CISL enables concise re-usable descriptions that one can easily extend to incorporate additional features of an architecture. In the following sections we elaborate on the details of CISL from the ground up: Section 3.2.1 introduces the CISL type system; in Section 3.2.2 we elaborate on how machine memory is described. Section 3.3 demonstrates how we use CISL to capture the encoding and semantics of instructions. Lastly, in Section 3.3.3 we discuss CISL's use of classes and mixins to show how we can write descriptions concisely that enable re-usability.

3.2.1 CISL Types

The basic type of data that any modern machine handles is the *bit*. As such, our model builds upon this fundamental concept by providing the type **`bit`**. Although useful in its own right, it is limited to representing only such things as the numeric values 1 or 0, booleans *true* or *false*, and *on* or *off*. To be truly useful, then, we must be able to organize bits into structures that are capable of representing components of a machine, in particular, storage locations. For this reason, the most heavily used type in our system is the **`bit`** array. A bit array is simply that: an array of elements of type **`bit`**. More generally, however, a bit array can be an array of bit arrays as illustrated in the following definition in Backus–Naur Form (BNF) notation,

$$
\begin{array}{lcl}
\mathrm{T} & \Rightarrow & \mathrm{B} \mid \mathrm{R} \\
\mathrm{B} & \Rightarrow & \texttt{bit} \\
\mathrm{B} & \Rightarrow & \mathrm{B}\texttt{[N]} \\
\mathrm{R} & \Rightarrow & \mathrm{real(N)} \\
\mathrm{N} & \Rightarrow & n, \text{ if } n \in \textit{Integer}
\end{array}
$$

Thus, a type T is either a bit or bit array of n elements (B) or a real type (R). Floating-point values are specified using the type *real* along with a specified length n. Currently, we assume that a *real* type refers to the abstract notion of a real number, not in any particular format or standard, although most present day machines adhere to the IEEE 754 floating-point standard.

Given the definition of bit arrays we can imagine any number of possible constructions that could represent data in a machine. For example, a bank of 16 32-bit registers can easily be represented by a bit array of size 16 where each entry contains an array of 32 bits, or more concisely, **`bit[16][32]`**. This is not only sufficient for specifying typical memory structures such as a register file or the JVM heap, but it is also useful for defining the type of an instruction word fetched from an instruction store, namely, **`bit[32]`** for a 32-bit instruction.

In addition to defining the structure of bit array types, a bit array also requires an interpretation. More specifically, an interpretation of a type allows us to map its value to a representation of bits. For our purposes, two *interpretation modifiers* are provided: *endianness* and *signedness*. In our model all bit arrays are associated with an endianness, either *big* or *little*, that indicate whether bit 0 is *most-significant* or *least-significant* respectively. This property allows us to reason about types in the native endianness of the machine as well as between different machines with opposite endianness. For example, the layout of bits on a big-endian bit array of size n on one machine is not the same as a little-endian bit array of size n on another machine even though they may represent the same value.

Another interpretation that an array of bits has is its signedness. In particular, does a given array of bits represent a *signed* or *unsigned* value? Since all machines generally support operations for both, providing such an interpretation is not only convenient but necessary for reasoning about equivalent operations between different architecture models. Unlike endianness, however, two bit arrays may be bit-by-bit identical but potentially represent different values from a signedness point of view. We also note here that we are assuming a *two's complement* representation. This is not a restriction of the system and could just as easily be *ones' complement*—indeed, most user mode instructions are unaware of the actual value representation of the underlying machinery. For example, a signed addition of two registers on a two's complement machine produces the same value result as a signed

addition on a ones' complement machine. That some programs depend on a specific binary representation is an issue in program semantics rather than of the instruction set.

Ultimately, these interpretation modifiers have implications on how we map compiler IR instructions to target instructions as well as how we prove their equivalence. While endianness provides us with directions as to how we extract the value from an array of bits, signedness indicates what the bits mean. Using this information we can check our model for correct operator usage when defining instruction semantics as well as check that operations on different machines are operating on the same kind of value.

To conclude our discussion on types, we show an example of the CISL syntax that allows us to introduce *named* types into the system. A named type definition associates a symbolic name or identifier with a particular type. This name can then be used in subsequent type definitions as we show in the following snippet of CISL:

```
type byte = little signed bit[8];
type word = little signed byte[4];
```

Here we declare a type **byte** that is 8 bits in length, is interpreted as a signed value, and its 0 bit is the least significant bit. We then define the type **word** that is an array of 4 bytes, is interpreted in its entirety as a signed value, with its 0 byte is the least significant byte.

3.2.2 Store

A store is specified in CISL using a *store declaration*. A store declaration contains a list of memory declarations describing the locations that are available on a particular machine. In CISL, there are two kinds of memories, *indexed* and *addressed*. A typical indexed memory is a register file. Accesses to indexed memories do not allow a computed index as they are are usually accessed by a hard-coded field of an instruction (e.g., register operand) or for a specific purpose (e.g., frame pointer). In Listing 3.3 we show an example of the store definition for the ARM architecture. The register file, **R**, is declared as an indexed memory of 16 elements of type **word_t**. One can also declare indexed memories that consist of a single location as shown in Listing 3.4 for the JVM stack pointer register (**SP**);

An addressed memory is usually larger and typically accessed with a computed address. For this reason, one only needs to specify the type used to index the memory. In our ARM example, the memory declaration for main memory, **M**, must be indexed with expression of type **address_t**. Because these kinds of memories often support several data sizes (e.g., byte and word) CISL allows one to define *data selectors* to indicate the type of the data that one can access as well as any alignment constraints. Alignment is specified as an integer following the data selector definition: **2** indicates that the access is aligned on two quantum or **4** is aligned on four quantum. One must specify a *quantum* data selector to indicate the smallest accessible unit of data; other selectors must be aggregations of the quantum. For example, ARM main memory is byte-oriented so the quantum is a byte, but it is also possible to access 16-bit half-words and 32-bit words of data, which we describe using the additional selectors **half** and **word**. The heap memory, **Heap**, for the JVM is defined similarly.

Listing 3.3: Store definition for the ARM

```
type address_t = little unsigned bit[32];
type byte_t    = little signed bit[8];
type half_t    = little signed bit[12];
type word_t    = little signed bit[32];

store Arm {
  address M[address_t] {
    quantum data byte_t byte;
    data half_t half 2;
    data word_t word 4;
  }

  indexed R[16][word_t] {
    alias PC = R[15];
  }
}
```

Listing 3.4: Store definition for the JVM

```
type baddress_t = big unsigned bit[32];
type bbyte_t    = big signed bit[8];
type bword_t    = big signed bit[32];

store JVM {
  address Heap[baddress_t] {
    quantum data bbyte_t byte;
```

```
    data word_t cell;
  }

  address Stack[baddress_t] {
    quantum data bword_t slot;
  }

  indexed SP[1][baddress_t] { }
}
```

In addition, memory declarations allow locations to be *aliased*. For example, the ARM program counter (**PC**) is the same as general purpose register 15. Although it is acceptable to reference **R[15]** in a semantic definition, using the name **PC** enhances the readability of the CISL description and can often be directly translated from an architecture manual.

Our CISL description treats condition codes differently than it does the memories we described above, because of their special control flow semantics. For example, many machines provide a compare instruction that compares two values, sets condition codes based on the comparison, and later uses those codes in a branch instruction, providing an implicit connection between the two. Because of this special control flow relationship, we define a set of *virtual* condition code registers that are used for specifying compare and branch style semantics. In particular, we provide condition code registers for the most common comparison operations, such as **==**, **!=**, and signed and unsigned **<**, **>**, **>=**, and **<=**. For use in a simulator, where one might need to save a processor status word, we can also compute carry, zero, negative, and overflow bits (or whatever the particular processor does), but the virtual bits are more useful for capturing compare and branch semantics. (Also, if they are computed lazily, they describe an important optimization for simulators.)

Here is an example of the condition code registers being set in our description of a PowerPC register compare instruction:

Listing 3.5: PowerPC codition codes

```
NE = R[RA] != R[RB]; UNE = (uword_t) R[RA] != (uword_t) R[RB];
EQ = R[RA] == R[RB]; UEQ = (uword_t) R[RA] == (uword_t) R[RB];
LT = R[RA] <  R[RB]; ULT = (uword_t) R[RA] <  (uword_t) R[RB];
GT = R[RA] >  R[RB]; UGT = (uword_t) R[RA] >  (uword_t) R[RB];
LE = R[RA] <= R[RB]; ULE = (uword_t) R[RA] <= (uword_t) R[RB];
```

```
GE = R[RA] >= R[RB]; UGE = (uword_t) R[RA] >= (uword_t) R[RB];
```

Note that CISL does not require the description author to indicate *how* to compute the actual bit that is stored in a virtual condition code register, just the comparison expression. We thus capture the semantics of condition codes in a uniform manner (for both signed and unsigned comparisons (e.g., NE, UNE)), which we can exploit in our search procedure since it matches IR and target instruction patterns. One then uses a condition code in a CISL **`if`** statement to specify the semantics of the PowerPC branch-if-greater-than instruction:

Listing 3.6: PowerPC codition codes

```
if (GT) {
  word_t delta = (BrDisp::0b00)!32;
  PC = PC + delta;
}
```

3.3 Describing Instructions

An instruction in CISL has two primary components: *encoding* and *semantics*. The encoding of an instruction represents its bit pattern. Typically, the encoding is divided into several *fields* that are used to identify the instruction type (e.g., opcode) or to provide parameters to the instruction. For example, bits 21-24, 26-27, and 4-6 shown in the encoding for the ARM ADD in Figure 3.1 are used to identify the instruction. The fields labeled by **`Rn`**, **`Rd`**, and **`Rm`** are used as parameters to the instruction semantics. We describe how CISL defines the encoding of an instruction in Section 3.3.1.

The semantics of an instruction specify what the instruction "does" when it is executed by the machine. The semantics operate on values defined by the encoding or memory locations declared by a CISL store. More formally, the semantics of an instruction is a function, *op*, mapping the machine state σ to a new machine state σ' or *op*: $\Sigma \rightarrow \Sigma$ (where $\sigma, \sigma' \in \Sigma$). How *op* accomplishes this mapping is called the *effect* of an instruction and is the primary mechanism we use to compare instructions on different machines for equivalence. The effect is also the basis for construction of functional simulators. CISL provides numer-

ous operators capable of describing the instruction semantics for modern architectures and compiler IR. We elaborate on the details of the semantics specification in Section 3.3.2.

3.3.1 Defining Instruction Encoding

As mentioned previously, to describe the encoding and semantics of an instruction CISL uses *object-oriented* techniques. In particular, classes are used to define instructions, variables describe the encoding structure of an instruction, and methods define semantics over those variables. We leverage these concepts for two distinct purposes: to provide a language that is familiar to those who have used an OO language previously (in fact, our syntax is very similar to languages such as Java and C#) and to exploit class inheritance and mixins to promote reuse. In this dissertation we discuss points of CISL that are relevant to generating instruction selectors; CISL supports the generation of other system tools including assemblers, disassemblers, and functional simulators [Moss et al., 2005, Richards et al., 2007]

Before we begin our discussion of the semantics, we show how to specifying the binary encoding. This is required by the semantics as instructions often use fields in the encoding, referred to as the *instruction parameters* (such as immediate values and memory addresses), directly or indirectly in expressing effects. An example of the encoding of the ARM *ADD logical shift left by immediate* instruction taken directly from the ARM Architectural Manual [Seal, 2000] is shown in Figure 3.1. This instruction is 32 bits long (in fact, because ARM is a fixed length instruction set all instruction words are 32 bits wide) and consists of several instruction fields.

Certain fields are fixed values that serve to identify the type of instruction such as the *opcode* field (bits 21 – 24 in the above figure). The values of other fields vary depending on the parameters of the instruction, for example, the destination register *Rd*. To accommodate these notions we define the encoding of an instruction using *variable declarations*

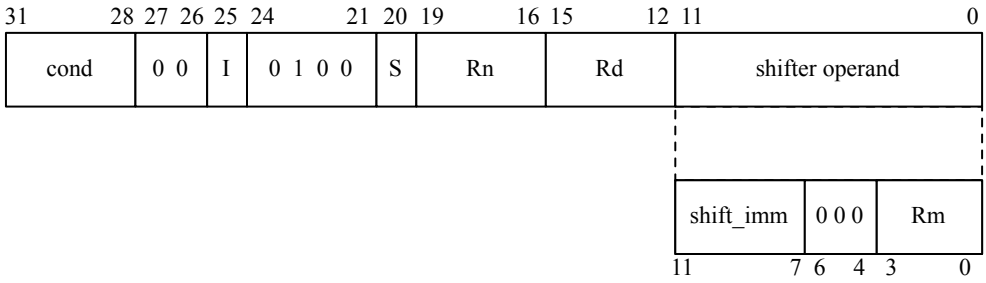


Figure 3.1: ARM ADD logical shift left by immediate

to indicate the structure of the encoding and *constraints* to indicate fixed values identifying an instruction.

A single variable declaration describes the overall form of an instruction word (i.e., length and endianness). A *reference declaration* is then used in conjunction with that variable to define new variables that refer to individual fields within the instruction word variable. For example, the following CISL class declaration defines the instruction word and fields for the ARM ADD instruction shown in Figure 3.7:

Listing 3.7: ARM ADD: Field Declarations

```
instruction class ADD_SHL_IMM {
    var bit[32] inst;
    var bit[4]  cond @ inst[28:31];
    var bit[4]  opcode @ inst[21:24];
    var bit[4]  Rn @ inst[16:19];
    var bit[4]  Rd @ inst[12:15];
    var bit[4]  Rm @ inst[0:3];
    var bit[5]  shift_imm @ inst[7:11];
}
```

This defines the instruction class **`ADD_SHL_IMM`** containing several variable declarations. In this example we are assuming the default endianness and signedness to be little-endian and unsigned respectively. We first define a 32-bit variable **`inst`** representing an ARM instruction word. The succeeding reference declarations refer to specific fields of bits within the variable **`inst`** using the *range* operator `:`. For example, **`cond`** defines the condition field that is 4 bits wide starting at bit 28 (because **`inst`** is little-endian this refers to bits 28

– 31), **opcode** is 4 bits wide starting at bit 21, **Rn** is 4 bits starting at bit 16, and **Rd** is 4 bits wide starting at bit 12.

As we mentioned previously, we fix certain fields to indicate the type of the instruction. To do this, we extend the instruction class definition in Listing 3.7 with a special method named *encode* whose body is a series of assignments that indicate the fixed values of these fields,

Listing 3.8: ARM ADD: Fixed Encoding

```
instruction class ADD_SHL_IMM {
    // variable declarations defined in previous listing
    fun encode() {
        opcode = 0b0100;
        inst[26:27] = 0b00;
    }
}
```

This example introduces several points of notation. First, comments begin with **//** and continue to the end of the line. Second, a method declaration begins with the **fun** keyword followed by a return type, the name of the method, and a list of formal arguments. The return type may be omitted if the method does not return a value as is the case for the **encode** declaration above. The body of the encode method consists of a list of assignment statements indicating the fixed values for particular variables or fields. For example, we set the **opcode** field to the binary value **0b0100** identifying this as an ARM ADD instruction. In addition to binary, CISL literals can also be specified in a variety of formats including decimal, hexadecimal, and octal.

In some cases a field may not have a name associated with it (i.e., no reference declaration) as we show by setting the value **0b00** for the field **inst[26:27]**. This expression refers to the two bit location within the **inst** array starting at offset 26 and ending at offset 27 (we elaborate on the complete set of operators when we discuss the semantics). In fact, reference declarations are really just syntactic sugar to provide more readable specifications. Indeed, we can specify the complete encoding directly using the range operator. Additional fixed encodings exist for this instruction but we omit them to conserve space.

We list example CISL descriptions of real instructions for real architectures and compiler IRs in Appendix A.

The CISL compiler treats the **`encode`** method differently from the semantic methods (discussed in the next section). In particular, the only operator that can be used in an encode method is assignment (**=**). We record these assignments as *field constraints* during processing and the CISL compiler expects back-end generator tools (e.g., GIST) to interpret them appropriately. For example, assemblers and instruction selector generation can use this information to generate machine code output; disassemblers and functional simulators can use field constraints to decode instructions for debuggers and simulated execution respectively.

In some instances an instruction may not have an encoding. This is particularly true for virtual instruction sets such as an IR used by a compiler. The omission of the encode method from an instruction specification indicates to the GIST system that the instruction cannot be emitted by a code generator in binary form (which makes sense for IR instructions).

Now that we have a firm understanding of the mechanics of variable declarations, references, and encoding we are in a position to discuss the instruction semantics, which defines what an instruction does. They specify the operations that take place (i.e., add, subtract) and the underlying stores that the instruction observes and updates (i.e., registers, memory). It is these two components that enable the GIST system to find a semantic mapping of an instruction on one machine into a sequence of instructions on a target machine if it exists.

3.3.2 Specifying Instruction Semantics

We designed the CISL language to describe instructions for real architectures and compiler IR. This is both novel and important for GIST in finding instruction selector patterns—because we specify IR and target semantics in the same language it is possible to compare them syntactically to determine equivalent semantics. In CISL, one describes the seman-

tics of an instruction using a combination of *expressions* and *statements*. An expression "produces" a value, where as a statement typically introduces a change in the machine state (i.e., storing a value to memory). Other statements exist to describe parallel assignments and exceptional conditions.

3.3.2.1 Variable Expressions

In a typical programming language, variables are the fundamental objects used to store and retrieve values. In CISL, we use three constructs in expressions to retrieve a value: memories, fields, and local variables. A memory reference in an expression "loads" a value from that memory declared in the store. We reference an indexed memory like an array in a conventional programming language:

```
R[5] + 12
```

This describes the addition of the integer literal **12** to the value stored in memory **R** at index **5**. If we declare an indexed store to have only a single element it can be referenced using its name alone, as shown in the following example where we subtract **4** from the stack pointer register (**SP**):

```
SP - 4
```

Addressed memories are different as they declare *data selectors* that are used to address different sizes of a stored value in memory (e.g., byte, half, word). The following example shows how ARM main memory **M** (declared in Listing 3.3) would be referenced:

(a) `M[R[Rd]]`

(b) `M.half[R[Rd]]`

(c) `M.word[R[Rd]]`

In example (a), we do not use a data selector—this indicates that we applied the default quantum data selector (**byte**). In (b) and (c) we use the data selectors **half** and **word** to

aggregate two and four consecutive bytes respectively. We can index addressed memories by any arbitrary expression (as long as it agrees in type as defined in the store). However, we can index indexed memories only by a literal or field. In each of the above cases, the reference to register **R** is indexed by the field **Rd**.

In addition to memories, it is possible to reference fields and local variables in semantic expressions. Unlike memory, however, it is not possible to assign a value to a field in a guarded assignment. This limitation makes sense for encoding fields because the meaning of an instruction dictates the semantics and its parameters, not the reverse. A local variable, however, is not part of the encoding, rather it is used as a named expression, resulting in semantic descriptions that are easier to read and maintain. A local variable is declared exactly like a field, however, it must provide an initialization expression:

(a) `var bit[32] x = M.word[Rd];`

(b) `x + 4`

In the above example, we declare the local variable **x** in (a) and in (b), we reference **x** in an add expression. Unlike fields, however, a local variable is accessible only in the method in which it is declared.

3.3.2.2 Expression Types

In CISL, the semantics of each operator have a well-defined meaning—this allows semantics to be matched syntactically. It is possible, however, for two operators of the same kind to not match because their operands have different types. For example, some machines (e.g., MIPS) may differentiate between unsigned and signed operations. One direction would be to define additional operators that syntactically indicate the difference in signedness, endianness, or length (as we have done for floating-point). This approach, however, can lead to semantic descriptions that are cluttered with too much type information. For this reason, the types of operators are inferred by the CISL compiler.

More precisely, operators in CISL are polymorphic in the type of their argument(s). For example, the CISL operator for addition, $+$, can be described by the *type schema* $\tau \times \tau \rightarrow \tau$. This means addition has two operands whose type is τ and produces a resulting value whose type is also τ. Alternatively, consider the type schema for the concatenation operator (::): $\tau_j \times \tau_k \rightarrow \tau_{j+k}$, where j and k indicate the bit lengths of each type of its operands. In this case, the resulting type has a bit length that is the addition of the lengths of its two operands. For example, the expression, $x :: y$, where x is a bit array of size 16 and y is a bit array of size 16, then the type of :: would be a bit array of size 32. These type schemas are used by the CISL type inference system to infer the types of all semantic expressions. Inferred types are then used by GIST to match operators precisely. For example, a match between two semantic expressions using $+$ will fail if they disagree in signedness or their bit lengths differ.

It is also possible to change the type of an expression by using the *cast* operator. In CISL, a cast can change only the signedness or endianness of an expression, but not the length—changing the bit length is accomplished using sign-extension or range. A typical case where a cast and sign-extension are both required is when an unsigned immediate, encoded in an instruction word, is added to a signed register. For example, the MIPS architecture provides an *add immediate unsigned* instruction. Because the CISL front-end compiler enforces strict type rules on an expression we must take care to ensure that we describe the semantics precisely as shown in the following example:

```
R[rd] = (little unsigned bit[32])R[rs] + (imm!32);
```

The immediate field, **`imm`**, is declared to be 16-bit unsigned. This is sign-extended to 32-bits and then added to register **`R[rs]`** which is appropriately cast to be **`unsigned`**. The resulting type of the addition is then **`little unsigned bit[32]`**. The type inference system automatically inserts a cast for assignment—in this case the result is automatically cast to **`signed`**.

3.3.2.3 Operators

CISL provides a significant number of operators to describe the semantics of an instruction. In Figure 3.2, we show all the CISL fixed-point binary and unary operators ordered by precedence. In addition to the operators found in mainstream programming languages (e.g., Java), several bit-level operations are provided such as population count and sign extension. CISL also supports a range of floating-point operators used to describe the semantics of floating-point instructions. A floating-point operator is the same as its fixed-point counterpart prefixed with a dot: ".". For example, floating-point addition is `.+` and multiplication is `.*`.

Logical Operators The logical operators, `==`, `<`, `>`, `<=`, `>=`, and `!=` are used to compare the *value* of two CISL bit structures of the same type. Although, it is possible for different sized bit arrays to represent the same logical value (i.e., `bit[32]` and `bit[16]`), real processors typically require their operands to be the same size. Because compiler IRs are targeting real instructions, they typically require the operands to be type compatible. If they are not, the front-end of the compiler uses conversion rules to promote operand(s) to the appropriate type.

The operators, `!`, `&&`, `||`, and `^^` require their operand(s) to evaluate to the CISL type *bit*. Because CISL is operating within the context of a single instruction, *short circuit evaluation* is not applicable as is the case in most high level programming languages. In other words, the operands to a logical operator are evaluated before the operation is applied.

Bit Resizing Operators We separate these operators from the bitwise operators as they are not common from a programming language standpoint. The bit resizing operators are used by CISL to resize a bit array. The range operator, :, is the most frequently used operator for describing instruction fields. It is used to access a range of bits within a larger bit array. For example, given the bit array `x` of size 32 that is little-endian and signed, then `x[0:5]` results in a bit array of size 6 that is also little-endian and signed. The operators `#` and `##` are used, along with an integer argument *n*, to resize a bit array to size *n* by padding 1 to

Operator	Operation	Notes
`==`	equals	
`<`,`>`,`<=`,`>=`	comparison operations	
`&&`	logical *and*	
`||`	logical *or*	
`^^`	logical *xor*	
`!`	logical *not*	
:	range	access a range of bits; used in an array index.
`#` *n*	front resize	resizes bit array by *n* adding 1 padding to front.
`#!` *n*	front resize zero	resizes bit array by *n* adding 0 padding to front.
`##` *n*	back resize	resizes bit array by *n* adding 1 padding to back.
`##!` *n*	back resize zero	resizes bit array by *n* adding 0 padding to back.
`!` *n*	sign-extend	sign-extend bit array by *n* bits
`!0` *n*	zero-extend	zero-extend bit array by *n* bits
`::`	concatenation	appends two bit arrays.
`&`	bitwise *and*	
`|`	bitwise *or*	
`^`	bitwise *xor*	
`~`	bitwise *not*	
`<<`,`>>`	left/right shift	
`!>>`	arithmetic right shift	
`<<<`,`>>>`	left/right rotate	
`+`,`-`,`/`,`*`	add/sub/div/mul	
`%`	modulo	
`-`	two's complement	unary negation.
`++`,`--`	increment/decrement	
`$1`,`$0`	population count	

Figure 3.2: CISL Operators

the front and back respectively. `#!` and `##!` are similar except they add 0 padding. The concatenation operator, `::`, is used to *append* two bit arrays with identical type modifiers (e.g., signedness and endianness) of size j and k to produce a bit array of size $j+k$. Lastly, CISL provides two operators, `!` and `!0`, to sign- or zero-extend a bit array respectively.

Bitwise Operators The operators `&`, `|`, and ~ (bitwise *and*, *or*, and *xor*) requires operands of the same type. The right-shift operator `>>` shifts the bits of an array to the right by the specified amount. The semantics of `>>` is different depending on the signedness of the array: if the array is signed, sign-preserving padding is used, otherwise the resulting array is zero padded. The left-shift operator `<<` shifts the bits of an array to the left by a specified amount padding the resulting array with zeros. The arithmetic shift-right operator, `!>>`, is similar to right-shift. However, the sign is ignored and is padded by zeros. Lastly, the rotate operators, `>>>` and `<<<`, rotate the bits in an array by a specified amount right and left respectively.

Arithmetic Operators The arithmetic operators `+`, `-`, `/`, `%`, and `*` all require their operands to be of the same type. The value of the increment expression `x++`, where `x` is a memory location, is `x+1`, and is translated by the CISL compiler into the equivalent effect `x=x+1`. This also occurs for expressions using the decrement operator. Additional semantics resulting from an arithmetic operation such as *overflow* or *carryout* are recorded by the semantic description by assigning to the CISL supported location. For example, to capture overflow resulting from signed addition one would assign to the built-in `OV` memory: `OV = x + y`.

Specialized Operators CISL includes two specialized operators for describing population count. `$1` and `$0` are used to count the number of 1's and 0's in a bit array respectively. Although it is possible to express population count using other operators and a bounded for-loop, it makes sense to fold specialized, yet common, semantics into a single operator as it makes matching faster and easier. It is certainly possible for a machine to introduce an operation that is not covered by the standard CISL operator set. These operations could

be defined using the primitive operators. However, depending on the operation, this may lead to complicated semantic descriptions that are difficult to match. For this reason, CISL has been designed to allow new operators to be defined. This introduces other problems in that the IR or target must be aware of the operator on both sides to allow matching to be successful or an axiom must be introduced to rewrite an operator into primitive operators. This does not make the approach less general. However, it requires additional work to map the IR to the target. The CISL descriptions that we have written so far have not required any new operators—all semantics have been easily expressed within the standard operator set.

3.3.2.4 Statements

A statement in CISL is used to describe a change in the state or control flow of the machine. There are four kinds of statements: *assignment*, *if-else*, *for-loop*, *all-loop*, and *throw*.

The `if` Statement

The *if-else* statement is used to describe instructions that have branching semantics. For example, the **`if_cmplt`** bytecode shown earlier in Listing 3.1 or the ARM's predicated instruction set. It is also possible to specify multiple branches using *else*:

```
if (C1) {
  S1
} else if (C2) {
  S2
} else if (C3) {
  S3
} else {
  S4
}
```

If the condition C_i is *true* then the corresponding statements S_i are executed. GIST translates an *if-else* statement into a sequence of conditional assignments—how this is accomplished is elaborated on in Chapter 6.

The `for` Statement

The *for-loop* is used to describe *bounded* iteration. and has the form:

```
for control in [start .. end] {
  S
}
```

The *for-loop* iterates from *start* to *end* assigning the current iteration value to the *control* "variable". The *control* variable is special in that a variable declaration is not required; its type is inferred by the *start* and *end* variables as well as how it is used with the body of the for-loop (*S*). For example, the ARM architecture provides an instruction to load and store multiple registers into memory. The *load multiple* instruction loads a set of registers that are specified in a *bit field* encoded in the instruction word. We can describe these semantics using a CISL *for-loop*:

```
for i in [0 .. 15] {
  if (reg_list[i]) {
    R[i] = M.word[R[Rn]];
    R[Rn] = R[Rn] + 4;
  }
}
```

In this example, we iterate from **`0`** to **`15`**, the registers available on the ARM (16) and the number of bits that encodes which registers to load (**`reg_list`**). Inside the loop we also have a conditional that determines if the bit for the specific register is set. If *true*, register **`i`** is loaded from memory at the location specified in register **`R[Rn]`**.

Because CISL loops are bounded it is possible to perform *loop unrolling* in the event GIST is unable to procure a match on the target for the entire loop. Consider an IR that includes an instruction that loops over an array of integers performing the same operation on each entry. It is very likely that this looping behavior is not supported directly on a real architecture. By unrolling the loop we can generate a straight-line implementation of the semantics for each iteration. In this form, it is more likely that GIST will be able to find a match.

The `all` Statement

The *all-loop* allows the definition of guarded effects that are to be *executed* in parallel and has the form:

```
all control in [start .. end] {
  S
}
```

all has a similar syntactic structure as the *for-loop*, but the semantics are radically different: *all* statement indicates that the guarded assignments in the body, *S*, are to be executed in parallel. The *control* variable is used to index a memory in *end-start*+1 different locations in parallel. The typical use case of an *all* is for describing *vector*-like instructions. For example, we use *all* in our description of the Synergistic Processing Element (SPE), a special-purpose vector unit, found on the Cell Broadband Engine [Kahle et al., 2005, Gschwind, 2006]. The SPE has a vector of 128 registers that are 128 bits wide. In CISL this is modeled with a indexed store named GPR containing 128 bit arrays that are 128 bits long. We model the *vector add* instruction as follows:

```
all i in [0 .. 3] {
  GPR[rt][i*32:i*32+31] = GPR[ra][i*32:i*32+31] +
                          GPR[rb][i*32:i*32+31];
}
```

The SPE supports single-instruction, multiple-data (SIMD) capability. Its SIMD instructions can operate in parallel on several consecutive sub-ranges of bits in multiple registers including sixteen 8-bit values, eight 16-bit values, four 32-bit values, and two double-precision floating point. The above example demonstrates how we would use the *all* statement to model the SPE 32-bit vector add instruction. The control variable, **i**, indicates which sub-range of bits are being operated on. The CISL variables **rt**, **ra**, and **rb** are specified by the encoding and indicate which 128-bit registers are involved in the operation. The sub-range of bits, within each register, is specified by the expression **i*32**, the first bit, and **i*32+31**, the end bit. The semantics can be more clearly observed if we "unroll" the *all* statement:

```
all {
  GPR[rt][0:31]   = GPR[ra][0:31]   + GPR[rb][0:31];
  GPR[rt][32:63]  = GPR[ra][32:63]  + GPR[rb][32:63];
  GPR[rt][64:95]  = GPR[ra][64:95]  + GPR[rb][64:95];
```

```
  GPR[rt][96:127] = GPR[ra][96:127] + GPR[rb][96:127];
}
```

This is shorthand to indicate that each guarded-effect is to be executed in parallel. It is equivalent to an *all* statement where both *start* and *end* are 1 and the *control* variable is not used.

The `throw` Statement

The throw statement is used to specify exceptional conditions that occur during instruction execution. This typically refers to semantics that are handled external to a user mode instruction. For example, an instruction may raise an exception if it attempts to divide by zero. An exceptional condition is not limited to real architectures; for example, the JVM *aaload* bytecode loads a reference from an array at a given index. If the index is not within the bounds of the array an *ArrayIndexOutOfBoundsException* is thrown. Both of these cases can be modeled by the CISL **`throw`** statement:

```
throw exception
```

An *exception* is a symbolic identifier naming the exception. Similar to stores, an *exception* identifier needs to be mapped to the proper exception on the target. CISL purposely does not describe the control flow semantics or memory structures used by an exception (i.e., exception vector (MIPS) or interrupt descriptor table (x86)) as the details of the semantics are different from machine to machine. However, the abstract semantics are often identical. The following example shows how one would describe the *aaload* instruction for the JVM:

```
// bounds check
var bit[32] ref   = S.slot[sp+4];
var bit[32] index = S.slot[sp];
if (index >= S.slot[arraysize(ref)] || index < 0) {
  throw ArrayIndexOutOfBoundsException;
}

// load array reference
S.slot[sp+4] = Heap.cell[arrayidx(ref,index)];
sp = sp + 4;
```

In this example we first perform a *bounds check*: we compare the size of the array to the index (both provided as arguments on the JVM run-time stack). It is important to note that we use a method call, **`arraysize`**, to determine the *length* of the array. Because the retrieval of an array's length on the JVM is implementation dependent, we can abstract the semantics into a method call and easily replace the implementation of **`arraysize`** depending on the specific JVM implementation. (In our experiments we describe two JVM implementations. However, it is easy to see that CISL is powerful enough to describe a generic JVM leaving implementation details to specific methods and/or classes and mixins.) If the index exceeds the size of the array the proper exception is thrown (e.g., raised); otherwise execution proceeds as normal and the value in the array at the given index is pushed on to the stack.

3.3.3 Classes and Mixins

As we demonstrated in the previous section, the core language of CISL (statements and expressions) is capable of describing the instruction set encoding and semantics of both compiler IR and real architectures. To allow CISL descriptions to be easily written, re-used, and extended we formally introduce *classes* and *mixins*. CISL provides a *class* declaration for grouping together related semantics that can be inherited by child classes. Inheritance facilitates concise descriptions of instruction fields and their decomposition. Consider the instruction formats found in the PowerPC. CISL class definitions readily capture these formats and their encodings, as shown in the following complete definition of the PowerPC **`addi`** instruction:

```
class Instruction {
  var bit[32] inst;
  var bit[6]  OPCD @ inst[0];
}

class DForm extends Instruction {
  var bit[ 5] RA     @ inst[11];
}

class DForm_RT_SI extends DForm {
```

```
  var bit[ 5] RT @ field1[0];
  var bit[16] SI @ field3[0];
}

instruction class addi extends DForm_RT_SI {
  fun encode() {
    OPCD = 0x0E;
  }

  fun effect() {
    GPR[RT] = GPR[RA] + SI!;
  }
}

instruction class addi_ra_zero extends DForm_RT_SI {
  fun encode() {
    OPCD = 0x0E;
    RA = 0;
  }

  fun effect() {
    GPR[RT] = SI!;
  }
}
```

In the PowerPC ISA, all instructions are 32 bits long and have a 6-bit opcode field. Therefore, our PowerPC description defines the base class **`Instruction`** with variable declarations for the instruction word (**`inst`**) and the 6-bit opcode field (**`OPCD`**). The **`Instruction`** class is *extended* by the class **`DForm`** to declare the encoding field **`RA`** used by the *DForm* format described in the PowerPC architecture manual [May et al., 1994]. This is further refined by the **`DForm_RT_SI`** class to describe the RT and SI fields.

One completes the definition of an instruction by indicating that its class is an *instruction class* and providing an **`effect`** method. In this example, we show two instruction class declarations for **`addi`** and **`addi_ra_zero`**. The former defines the PowerPC **`addi`** instruction without any constraints (except the **`OPCD`**) and the latter shows a variant where the **`RT`** field has been constrained to be **`0`** in the encode method. In this case, the semantics are that the immediate field (**`SI`**) is assigned directly to the register **`RT`**. Although this example uses inheritance to facilitate sharing of encoding information it is also possible to

reuse semantic methods that can be "called" from the effect method. For details see the CISL description listings in Appendix A.

Although classes are sufficient for describing much of the information related to instructions, we often encounter situations where properties are shared in a manner that is not easily captured by single inheritance. Consider the *shifter operand* addressing mode used by the ARM [Seal, 2000] data-processing instructions. There are 11 different forms represented by this addressing mode, used by 16 different instructions, leading to a total of 176 unique instruction formats and semantics (not to mention the 16 different condition field opcodes). To define the format for these instructions using only single inheritance would require 11 classes for each of the shifter operands and 16 classes that represent each of the data processing instructions, duplicated 11 times, inheriting a different shifter operand class with each. This quickly leads to an unwieldy description that is hard to read, understand, and maintain. What we require is the notion of attaching variables and methods to classes without disrupting the class hierarchy—for this we use *mixins*.

A mixin is similar in content to a class but is *used* by classes rather than extended. When a class uses a mixin, the declarations in the mixin become available to that class. In addition, it is possible (and common) for a class to use more than one mixin. This does not mean, however, that all the declarations in the used mixins become available simultaneously. Consider the following simple mixin use case: a class declaration *K* uses mixins *X* and *Y*. Using both mixins results in two unique class definitions: a class *K* that uses mixin *X* and a class *K* that uses mixin *Y*. In effect, a mixin can be viewed as a kind of *class constructor*. The following shows how we use mixins to describe the encoding for a SPARC *ldsb* instruction with two different encodings for the *i* field:

```
mixin reg_access {
  var bit i @ inst[13];
}

mixin direct_address extends reg_access {
  fun encode() { i = 0; }
}
```

```
mixin offset_access extends reg_access {
  fun encode() { i = 1; }
}

instruction class LDSB extends Format3
        uses direct_address, offset_access {
  fun encode() { op3 = 0b001001; }
}
```

We first define the mixin **reg_access** that provides the variable reference declaration for the **i** field (which incidentally resides at offset 13 in the instruction word **inst**). This mixin is then inherited by two sub-mixins: **direct_address** and **offset_access**. These two mixins define different encoding values for the **i** field (0 and 1 respectively). The **LDSB** instruction class then uses *both* mixins for two unique (non-conflicting) encodings defined over the *i* field.

This demonstrates the use of mixins for defining simple encoding alternatives. The real power of mixins comes into fruition when they are used to define "cross-cutting" semantics that permeate instruction definitions. For example, the ARM architecture defines a *predicated* instruction set—all instructions are conditionally executed. There are 4 condition code registers on the ARM that can be combined in a variety of ways to yield 13 different combinations used to conditionally execute an instruction. Using class inheritance alone would require each ARM instruction to have 13 different alternatives specifying a slight variation for the predication (not to mention the number of alternatives resulting from the shifter-operand addressing mode mentioned previously). For this reason, we use mixins to define the semantics for each condition in isolation:

```
mixin Eq {
  fun encode() { cond = 0b0000; }
  fun bit condPassed() { return EQ; }
}

mixin Ne {
  fun encode() { cond = 0b0001; }
  fun bit condPassed() { return NE; }
}
```

```
mixin Ge {
  fun encode() { cond = 0b1010; }
  fun bit condPassed() { return GE; }
}

mixin Gt {
  fun encode() { cond = 0b1100; }
  fun bit condPassed() { return GT; }
}

mixin Le {
  fun encode() { cond = 0b1101; }
  fun bit condPassed() { return LE; }
}

mixin Al {
  fun encode() { cond = 0b1110; }
  fun bit condPassed() { return 1; }
}
```

Here we show 6 of the mixin definitions (to conserve space) for the ARM conditions. Each mixin contains a **`condPassed`** method that describes the condition using the CISL condition code registers. A single bit is returned to indicate if the condition passed or failed. It is important to note that each mixin declares a method of the same name (e.g., **`condPassed`**). This allows the CISL compiler to generate multiple unique versions of the semantics each referring to a different **`condPassed`** method. This is a similar use case as we described above for the encoding mixins.

The other use case for mixins is when multiple cross-cutting semantics are used within the same instruction. For example, in addition to the predicated execution of each instruction, the ARM data processing instructions define 8 shifter-operand addressing modes. The combination of condition and shifter-operand yields a cross-product of unique instruction semantics. For this reason, when multiple mixin "groups" are used in combination to define an instruction in CISL, a cross-product of unique instruction definitions are generated by the CISL compiler. Consider the following CISL class declarations using the ARM condition mixins and shifter-operand mixins:

```
class CondInstruction extends Instruction
        uses Eq, Ne, Ge, Lt, Gt, Le, Al {
var big signed bit[4] cond @ inst[0:3];
}

class DataProcessingInstruction extends CondInstruction
        uses ImmediateShifterOperand, RegisterShifterOperand,
             LogicalShiftRightImm, LogicalShiftLeftImm,
             LogicalShiftRightReg, LogicalShiftLeftReg,
             ArithmeticShiftRightImm, ArithmeticShiftRightReg {
var big signed bit[4] op @ inst[7:10];
var big signed bit[1] s  @ inst[11:11];
var rinx_t rn @ inst[12:15];
var rinx_t rd @ inst[16:19];
}

// Arithmetic instructions
instruction class mov extends DataProcessingInstruction {
fun encode() {
op = 0b1101;
inst[4:5] = 0b00;
}

fun effect() {
   if (condPassed()) {
       R[rd] = shifter_operand();
   }
}
}
```

Each of the shifter-operand mixins define a method named **shifter_operand** that define the semantics for the addressing mode. The **CondInstruction** class uses the subset of condition mixins we illustrated above. The **DataProcessingInstruction** class extends the **CondInstruction** class and uses a subset of shifter operand mixins (see Appendix A for the complete definition). At this point, we have not made reference to the methods used in either of these mixin groups. Lastly, we define the **mov** instruction by inheriting **DataProcessingInstruction** and defining the **effect** method. The effect method makes use of both the **condPassed** and **shifter_operand** methods. Because the instruction class declaration for the ARM *mov* instruction references two different mixin methods, unique semantics for the cross-product of the mixins are generated.

This means, that for each of the 13 condition mixins and 11 shifter operand mixins a total of 13×11 or 143 unique instruction classes are generated by the CISL language. This drastically reduces the effort required to describe an architecture. In addition, it is easy to make changes to classes or mixins to define architecture variants for exploratory research.

3.4 Summary

In this chapter we introduced the CISL language. We provided an overview of the CISL type system and emphasized its role in semantic matching. We presented the CISL syntax and semantics in detail and demonstrated its *unique unique ability for describing the instruction semantics of compiler IR and real architectures*. We formally described the CISL operator set and the variety of statements that can be used to specify state change in a machine. To conclude our discussion we show in Figure 3.3 and Figure 3.4 snippets of CISL "code" describing instructions for a number of architectures and compiler IRs respectively.

```
instruction class Mvn extends DataProcessingInstruction {
  fun encode() {
    op = 0b1111;
    inst[4:5] = 0b00;
  }
  fun effect() {
    var bit passed = condPassed();
    if (passed) {
      R[rd] = ~shifter_operand();
      setNZ(R[rd]);
      setCarry(shifter_carry_out());
    }
  }
}
```

ARM

```
instruction class AND_R extends R_Format {
  fun encode() {
    op = 0;
    shamt = 0;
    funct = 0b100100;
  }
  fun effect() {
    R[rd] = R[rs] & R[rt];
  }
}
```

MIPS

```
instruction class ISZ extends Inst {
  fun encode() {
    op = 0b010;
  }

  fun effect() {
    M.val[effAddr(offs)] = M.val[effAddr(offs)]+1;
    if (M.val[direct(effAddr(offs))] == 0) {
      PC[0] = PC[0] + 1;
    }
  }
}
```

PDP8

```
instruction class lwz extends DForm_RT_RA_D
                      uses LoadWord, LoadWordZero {
  fun encode() {
    OPCD = 32;
  }

  fun effect() {
    R[RT] = loadWord();
  }
}
```

PowerPC

```
instruction class ADD_imm extends Format3b {
  fun encode() {
    op3 = 0b000000;
    op = 2;
  }
  fun effect() {
    var word_t const = (type word_t) (simm13!32);
    W[rd] = W[rs1] + const;
  }
}
```

SPARC v8

```
instruction class mpyu extends RR_Format {
  fun encode() {
    OP = 0b01111001100;
  }
  fun effect() {
    var word_vec_t rt = (type word_vec_t) GPR[RT];
    var word_vec_t ra = (type word_vec_t) GPR[RA];
    var word_vec_t rb = (type word_vec_t) GPR[RB];
    var byte_t i;
    all i in [0 .. 3] {
      rt[i] = (rb[i] & 0xFFFF) * (ra[i] & 0xFFFF);
    }
    GPR[RT] = (type qword_t) rt;
  }
}
```

CBE SPU

```
instruction class MOV_R2RM_32 extends OneByteOpcode
  uses RegEAX, RegECX, RegEDX, RegEBX, RegESP, RegEBP, RegESI, RegEDI,
    ModRM32_EAX_EA, ModRM32_ECX_EA, ModRM32_EDX_EA, ModRM32_EBX_EA,
    ModRM32_SIB_EA, ModRM32_Disp32_EA, ModRM32_ESI_EA, ModRM32_EDI_EA,
    ModRM32_EAX_Disp8_EA, ModRM32_ECX_Disp8_EA, ModRM32_EDX_Disp8_EA, ModRM32_EBX_Disp8_EA,
    ModRM32_SIB_Disp8_EA, ModRM32_EBP_Disp8_EA, ModRM32_ESI_Disp8_EA, ModRM32_EDI_Disp8_EA,
    ModRM32_EAX_Disp32_EA, ModRM32_ECX_Disp32_EA, ModRM32_EDX_Disp32_EA, ModRM32_EBX_Disp32_EA,
    ModRM32_SIB_Disp32_EA, ModRM32_EBP_Disp32_EA, ModRM32_ESI_Disp32_EA, ModRM32_EDI_Disp32_EA,
    ModRM32_EAX, ModRM32_ECX, ModRM32_EDX, ModRM32_EBX, ModRM32_ESP, ModRM32_EBP, ModRM32_ESI,
    ModRM32_EDI
{
  var byte_t mod_rm = M.byte[fetchByte()];
  var little unsigned bit[2] mod @ mod_rm[7:6];
  var little unsigned bit[3] reg @ mod_rm[5:3];
  var little unsigned bit[3] r_m @ mod_rm[2:0];
  fun encode() {
    D_FLAG = 1;
    inst   = 0x89;
  }
  fun effect() {
    setLoc(getDWordReg());
  }
}
```

Intel IA-32

Figure 3.3: Architecture Instructions in CISL

```
instruction class getfield_boolean extends ByteCode {
    var word_t fieldOffset;

    fun encode() {
        op = 0xb4;
    }

    fun effect() {
        T[0] = S.slot[direct(spTopOffset)];
        S.slot[direct(spTopOffset)] =
          H.byte[fieldRef(T[0], fieldOffset)] !0 32;
    }
}
```

Jikes RVM Implicit

```
instruction class getfield_boolean extends ByteCode {
    var word_t fieldOffset;

    fun encode() {
        op = 0xb4;
    }

    fun effect() {
        T[0] = S.slot[SP];
        S.slot[SP] =
          H.byte[fieldRef(T[0], fieldOffset)] !0 32;
    }
}
```

Jikes RVM Explicit

```
 instruction class subacc {
   var addr_t offset;

   fun encode() {

   }

   fun effect() {
     ACC[0] = M.val[direct(offset)] - ACC[0];
   }
 }
```

PQCC IR

```
instruction class ADDI4_C {
  var addr_t src1;
  var word_t const;
  var addr_t dst;

  fun effect() {
  T[dst] = T[src1] + const;
  }
}
```

LCC IR

Figure 3.4: IR Instructions in CISL

CHAPTER 4

SUPPORTING INSTRUCTION SELECTOR DESIGN

4.1 Introduction

In the previous chapter we presented CISL, a language that is capable of specifying the semantics of instructions for compiler IR and target instruction sets. Describing IR instructions independently of a compiler framework has a number of advantages. At the very least, it clearly documents their meaning, perhaps to a much greater level of detail than what is provided with some compilers. More importantly, however, formal specification allows the design of the compiler and the implementation of the IR on a target machine to evolve independently. For example, adding a new IR operation to the system requires only adding an instruction class in a CISL description (of course, the compiler front-end must be capable of producing the new IR).

In addition to an IR description, it is important to support a range of design decisions that enable generation of instruction selector patterns with desired characteristics. Such design decisions can have a significant impact on the performance of the instruction selector and on the generated code. For example, the Jikes RVM Baseline compiler [Alpern et al., 1999a,b, 2000b, 2005] ("Baseline" hereafter) translates bytecodes directly into machine code for a target machine. The initial PowerPC [May et al., 1994] implementation targeted the JVM operand stack [Lindholm and Yellin, 1999] explicitly, reserving a target register for the JVM stack pointer and generating instructions to manage the stack pointer based on the semantics of each bytecode. An improved version removed the explicit stack pointer and used direct offsets into the stack frame, thereby reducing the number of target instructions and improving the speed of the code. The important observation is that the meaning

of the bytecodes and target instructions did not change, but the design decisions resulted in different instruction selector patterns.

In this work, we currently recognize two important categories of instruction selector design patterns: IR store mapping and instruction selector state. An IR description specifies IR-level memory spaces that are convenient for expressing the semantics of its instructions. One needs to map these memory spaces to memories on the target that target instructions can access. Without this mapping one cannot match IR to target trees. We use instruction selector state to take into account compile-time information that impacts the kinds of code generator patterns we find. GIST supports these design patterns with a store mapping and an instruction selector state specification, respectively. The rest of this chapter elaborates on how we describe instruction selector state and store mapping and how they provide a novel separation between compiler design decisions and GIST's ability to take advantage of these choices to discover target sequences based on these decisions.

4.2 Instruction Selector State

While GIST focuses on a functional view of instruction selector patterns, i.e., that they are pure functions from IR instructions to target instruction sequences, we have observed that it is helpful to have some amount of instruction selector state visible to GIST. Above we mentioned the case of supporting an *implicit* stack pointer (top of stack). In the JikesRVM Baseline Compiler targeting the PowerPC, the implicit stack pointer is modeled as a Java variable named **`spTopOffset`** of type **`int`**:

```
// current offset of the sp from fp
public int spTopOffset;
```

Target sequences are generated with an "emit" method for each JVM bytecode defined by the JVM specification. Consider the Java code for the emit method corresponding to the **`isub`** bytecode:

```
protected final void emit_isub() {
 popInt(T0);
```

```
    popInt(T1);
    asm.emitSUBFC(T2, T0, T1);
    pushInt(T2);
  }
```

This bytecode "pops" the top two integer values off the JVM working stack into temporary scratch registers **T0** and **T1** respectively (**T0** and **T1** are assigned to fixed registers on the PowerPC). The PowerPC instruction **SUBFC** is then used to perform the subtraction, storing the result into location **T2** (also a fixed register), and then subsequently "pushed" onto the stack. An examination of the Java code for **popInt** and **pushInt** reveal the generation of load and store instructions that use and manipulate the compile-time state variable **spTopOffset**:

```
  private void popInt(int reg) {
    asm.emitLInt(reg, spTopOffset +
                 BYTES_IN_STACKSLOT - BYTES_IN_INT, FP);
    spTopOffset += BYTES_IN_STACKSLOT;
  }

  private void pushInt(int reg) {
    asm.emitSTW(reg, spTopOffset - BYTES_IN_INT, FP);
    spTopOffset -= BYTES_IN_STACKSLOT;
  }
```

The Java code generating the PowerPC load and store instructions use **spTopOffset** to refer to the fixed offset into the frame. Lastly, in each of these cases, the offset is incremented/decremented to reflect the current top of stack location. Other parameters (e.g., **BYTES_IN_STACKSLOT**, **BYTES_IN_INT**) refer to hard-coded values that are assigned specific values according to the target architecture.

In CISL, we want to model the implicit stack pointer as instruction selector state, updated by each Baseline IR instruction according to the JVM specification's model of the JVM working stack. Instruction selector state *looks* like processor (IR) state, modeled in CISL using CISL types, variables, etc. For example, the implicit stack pointer used by Baseline targeting PowerPC is defined by the CISL variable:

```
  var word_t spTopOffset;
```

This definition is used like other instruction fields to describe the semantics for the JVM *isub* bytecode for implicit Baseline:

Listing 4.1: `isub` Bytecode Example

```
S[spTopOffset] = S[spTopOffset] -
                 S[spTopOffset+1];
```

As previously mentioned, **`spTopOffset`** is used to index the stack rather than using a dedicated stack pointer register. This variable maps directly to the Java integer defined in the Baseline code. For this reason, it is unnecessary to to model an update to **`spTopOffset`** in CISL as it is updated by the Baseline Java code. The adapter (described in Chapter 7) used to translate GIST selector patterns into Java code for Baseline ensures that the generated emit methods perform the proper updating of **`spTopOffset`**. The resulting pattern disovered by GIST and generated by the Baseline PowerPC adapter is:

```
protected final void emit_isub() {
  asm.emitLWZ(T0, 4+spTopOffset, 1);
  asm.emitLWZ(T1, spTopOffset, 1);
  asm.emitSUBFC(T2, T1, T0);
  asm.emitSTW(T2, 4+spTopOffset, 1);
  spTopOffset += 4;
}
```

Because the adapter is translating instruction selector patterns generated by GIST, it is not necessary to use helper methods (e.g., **`popInt`**, **`pushInt`**) that aid the human implementors in defining hand-written emit procedures. At the same time, adapters are flexible enough to take advantage of pre-existing compiler infrastructure to generate code that can easily "plug in" to the existing instruction selector component. In this case, we use the scratch register definitions defined by JikesRVM in place of PowerPC registers[1].

We use a similar technique to model implicit Baseline's allocation of JVM local variables, which are allocated slots in the stack frame. Specific bytecodes (e.g., *aload*) are used

[1] In the JikesRVM PowerPC Baseline compiler **`T0`** through **`T6`** are defined as Java integers that are assigned to general purpose registers R3 to R9 reserved as scratch. The adapter could have easily substituted literal constants in place of these named variables; however, for debugging purposes we chose the symbolic temporaries.

to load and store values to the local variables. Baseline tries to allocate PowerPC registers to as many local variables as possible. The *location* of each of the local variables is maintained in Java code by an array of integers (**localFixedLocations**). A positive value indicates a register and a negative value represents an offset in the frame. Both cases lead to different target sequences so they must be modeled explicitly. We model the local variable array with an indexed store:

```
indexed LL[32][address_t]
```

The location is a parameter to the bytecode and is modeled differently for the register and frame offset cases. Consider the **ret** bytecode, which branches to the address contained by a local variable indicated as part of the bytecode. We model **ret** using two CISL instruction classes illustrated in Listing 4.2. **ret_reg** defines the register semantics by loading the **PC** with the value stored in **LL**. We use a store map rule (described in Section 4.3) to implement **LL** using the general purpose registers on the PowerPC. **ret_mem** is the alternate semantics that sets the **PC** with a value stored at an offset in the frame. Both instructions define a **location** variable—the first is used as a register index, the second as a frame offset. Note that both definitions are required because Baseline has two distinct implementations of the **ret** bytecode. This is not a deficiency of CISL, rather it is how Baseline implements the JVM specification, and it illustrates CISL's ability to describe complicated compiler state.

Listing 4.2: `ret` Bytecode Example

```
instruction class ret_reg extends ByteCode {
  var big unsigned bit[5] location;
  fun effect() {  PC[0] = LL[location];  }
}

instruction class ret_mem extends ByteCode {
  var word_t location;
  fun effect() {  PC[0] = S.slot[location-4];  }
}
```

It is important to note that although the CISL language is capable of describing external instruction selector state, it does not require special syntax or notation—it looks just like

ordinary CISL code. The connection between CISL definitions of selector state and the internals of the compiler are made externally by the adapters. This separation of concerns is important because it does not restrict the re-use of the IR description, allowing it to be used in other compiler frameworks. Because the state is typically a variable or array name in the code generator implementation (e.g., **`location`**, **`LL`**) the adapter often performs little or no work during the process of generating the instruction selector component.

4.3 Mapping the Store

In addition to instruction selector state decisions described in the last section, we must consider how we implement IR memories on the target. Similar to decisions regarding instruction selector state, IR memory implementation can impact the possible target sequences that GIST might discover for the IR. The operators of CISL have well defined meanings that allow us to match operator to operator. Memories, however, come in many shapes and sizes. Because of that, mapping from the IR's view of memory onto a target can also take multiple forms, each having the potential to impact the overall performance of generated code. For this reason, it is not only important that we are able to specify a *store mapping* from a code generator design perspective, but necessary if we are going to be matching IR and target semantics to determine equivalence. Further, it's an important design decision—so it is not obvious we want to automate it entirely.

Consider the case of generating an instruction selector for Baseline targeting the ARM architecture.[2] The JVM has a memory for the stack and a memory for the heap, and the two are disjoint. The ARM, however, has only a single flat memory space and a set of general purpose registers. Mapping instructions from the JVM to the ARM is thus complicated by the fact that values may be retrieved from different locations yet the semantics of these instructions have the same overall effect. In the simplest case, we designate two (implicitly) disjoint regions of ARM memory to be reserved for the JVM heap and stack. In addition,

[2] A platform that is *not* currently supported in Jikes RVM.

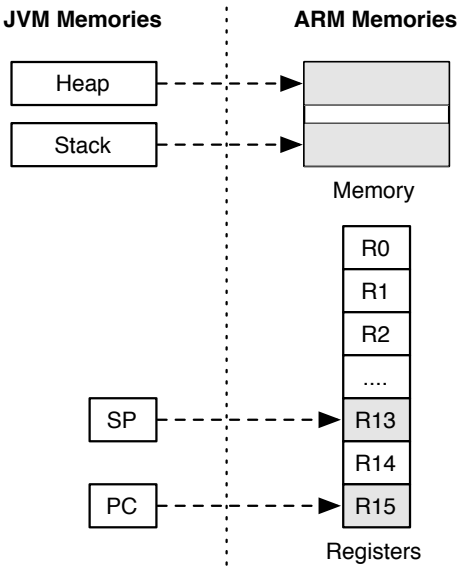


Figure 4.1: JVM to ARM Store Mapping

we can specify the JVM stack pointer, **`SP`**, as mapped onto register 13 on the ARM (in fact, this is an ARM convention). Alternatively, we could reserve a set of *n* registers that correspond to the top *n* locations on the stack for faster access. When the number of values on the stack is greater than *n*, values would be *spilled* to memory at some well known location. Figure 4.1 illustrates some of the memory mappings we use for the simple case.

The distinguishing feature of store mapping is that a mapped instruction has the same effect on the same values, but the locations of those values and how the instruction retrieves them are different. Thus, before we can determine the sequence of instructions on the ARM that is semantically equivalent to the JVM **`isub`** instruction (Listing 4.1), we must first view the semantic description of that **`isub`** in terms of the memories of the ARM. In other words, we must have a correspondence or mapping between the specific locations used by the JVM **`isub`** instruction and a subset of locations found on the ARM.

To enable the GIST user to control the mapping between the source IR and target machine stores, GIST supports the explicit specification of which memory locations defined by the target machine implement each memory of the source IR. This specification is called a *store map schema* and represents a specific implementation choice by the compiler im-

plementor. A store map schema allows considerable flexibility for the implementor of an instruction selector. For example, one could specify a simple mapping quickly to start producing a working code generator immediately, whereas a more complicated mapping may require more effort but yield more efficient generated code.

Listing 4.3: Explicit Baseline to PowerPC

```
map baseline to ppc {
  S.slot[T[$x]+$y] => M.word[R[`$x+3`]+$y] if $x <= 7;
  S.slot[SP+$x] => M.word[R[8]-`$x*4`];
  LL[$x] => R[$x];
  ...
}
```

Listing 4.4: Implicit Baseline to PowerPC

```
map baseline to ppc {
  S.slot[T[$x]+$y] => M.word[R[`$x+3`]+$y] if $x <= 7;
  S.slot[spTopOffset+$x] => M.word[spTopOffset-`$x*4`];
  LL[$x] => R[$x];
  ...
}
```

Listing 4.5: Top Three Register Baseline to PowerPC

```
map baseline to ppc {
  S.slot[T[$x]+$y] => M.word[R[`$x+3`]+\$y] if $x <= 7;
  S.slot[SP+$x] => M.word[SP-`($x-3)*4`] if $x > 2;
  S.slot[SP+$x] => R[`$x`] if $x <= 2;
  LL[$x] => R[$x];
  ...
}
```

A store map schema is a collection of rules that rewrite CISL store expressions that reference IR memories in an IR description into store expressions that reference target memories. In addition, rules can be applied conditionally based on matched values in the IR semantics, and they support an evaluation mechanism to generate relevant offsets on the target. The general form of a store map schema is captured by the following grammar:

schema	→	**map** ir-name **to** isa-name { *rules* }
rules	→	*rule* **;** *rules*
	|	λ
rule	→	*ir-cisl* **=>** *isa-cisl pred*
pred	→	**if** *condition*

A *schema* defines a mapping from IR (ir-name) memories to memory spaces used by an ISA (isa-name) using a list of *rules* that are separated by **;**. A rule has a left-hand side (*ir-cisl*), followed by **=>**, then a right-hand side (*isa-cisl*). The left-hand side defines patterns written in CISL that match CISL memory references found in the IR description. The right-hand side (also written in CISL) represents ISA memory reference patterns that are generated with a successful match. Rules are matched in the order they are defined; a successful match rewrites the IR CISL tree with the target memory patterns (isa-cisl).

In addition to CISL patterns, expressions in a store map schema can use pattern variables to match subtrees. A pattern variable looks like a regular CISL identifier, but is prefixed with **$**. If the right-hand side of a rule uses a variable **$x**, then the left-hand side must also use the variable **$x**. Variables can be used to map IR subtrees to ISA subtrees, or they can be used in an *evaluation expression* to generate new trees. An evaluation expression is specified between two back quotes: **`** *evalexp* **`** and can occur only as part of the right-hand side ISA pattern. The *evalexp* is an expression that is evaluated when a match is successful, and its result is used to construct the ISA memory reference. For example, the simple rule: **T[$x]+4 => R[`$x+3`]**, contains the expression (**`$x+3`**) that is evaluated if a match is successful. If the IR description contained the CISL code **T[3]+4**, then **$x** would bind to **3** and the expression **3+3** would be evaluated to produce **R[6]** for the target memory. The use of evaluation expressions occur most often when we wish to map IR locations to a pre-determined range of target locations or for computing offsets that depend on particular memory widths (i.e., references in words to bytes).

Lastly, a rule can be conditionally applied as indicated by *pred* in the store map grammar above. The *condition* is a simple boolean function used to determine if a rule is applicable with respect to a successful match. The use of a rule predicate occurs most often when we want a rule to apply only to a range of store locations. For example, if we wanted to constrain the rule we defined in the previous paragraph to target locations that are less than 7, we would specify the rule as: `T[$x]+4 => R[`$x+3`] if $x <= 7`. Simple comparisons can be combined to form complicated conditions using `&&` , `||`, and `!`.

In Listings 4.4, 4.3, and 4.5 we show three different store map schemas that translate memories from the JVM bytecode to PowerPC memories. Each captures a different implementation decision for mapping the JVM stack to the PowerPC target. The first rule is the same across all implementations and maps a stack access on Baseline to a memory access on PowerPC. The reference to the Baseline memory `T[$x]` in this rule refers to eight local variables defined in Baseline's code as we mentioned previously. We observe that this mapping has a guard, `if $x <= 7`, to restrict the value of the pattern variable `$x`, and the expression `` `$x+3` `` (evaluated during the mapping process). Thus, the reference `T[4]` in Baseline corresponds to register `R[7]` on PowerPC, whereas a reference to `T[5]` would not apply in this case and would result in a schema error during the mapping process.

The second rule defines three different implementations of the stack pointer for accessing the JVM working stack. The rule in Figure 4.3 for *explicit* Baseline maps a stack access using the stack pointer register `SP` plus an offset (`$x`) to the reserved register `R[8]` subtracting the expression `` `$x*4` ``. As mentioned above, the JVM defines offsets in terms of cells whereas the PowerPC expects bytes. Thus, this rule can easily translate the CISL semantics `SP+2` into `R[8]-8`. In Figure 4.4 the second rule for *implicit* Baseline maps a stack access using the compiler state variable `spTopOffset`, carefully adjusting the offset for byte access. Lastly, in Figure 4.5, the second and third rule define a mapping scheme that observes an implementation where the top three values on the stack are stored in registers `R[0]`, `R[1]`, and `R[2]`. In this case, we access the stack memory when the

offset (**$x**) is greater than 2, otherwise the offset is used to index the appropriate register. The last rule in each of these examples show a mapping from the local variable memory **LL** to any general purpose register on the PowerPC.

4.4 Summary

The emphasis of this work is on automating instruction selector construction. However, full automation can lead to fixed design decisions, making it diffcult to use in other compiler frameworks and limiting its use of target resources that can take advantage of clever performance enhancements. The lack of emphasis on offering design choice in prior work has limited the practicality, portability, and use of those systems. In this chapter we proposed the idea that supporting a degree of design flexibility, at the expense of full automation, is important to improve the portability of the approach and provides a mechanism for generating instruction selectors that take better advantage of the compiler implementation and target machine. We identified two important design patterns highlighting implementation decisions that emphasize instruction selector state and the representation of IR memories on the target. Instruction selector state is specified in a compiler independent fashion using CISL types and field declarations. In CISL, these fields look like other fields and are viewed as instruction parameters. This approach is transparent to GIST and its procedure for discovering target sequences. Adapters are used to bridge the state fields into compiler specific code (we present examples in Chapter 7). The representation of IR memories on the target is described by a set of rules called a store map schema. These rules define a specific implementation of IR memories in the target memory space. They conditionally match IR memory reference patterns and produce reference expressions using target memories. Store mapping leverages target resources to improve the performance and portability of generated code, whereas selector state improves performance by taking advantage of the implementation of the instruction selector.

CHAPTER 5

CANONICALIZATION

5.1 Introduction

GIST first converts CISL descriptions to an annotated abstract syntax tree (AST) representation. To give description writers flexibility and greater convenience, the language does not restrict the description of instruction semantics. As a result it is possible to specify identical semantics in different but algebraically equivalent ways. Of course, this requires the search for instruction selector patterns to consider different forms of the same semantics—making our problem significantly more difficult.

Prior work solved this by applying semantics-preserving transformations on trees during the search for instruction selector patterns. For example, if the search did not find a match for the source expression $a+b$, it would apply the commutativity of addition axiom to get $b+a$ and try to match that. This approach introduces extra work during search. Because rules such as commutativity and associativity are non-terminating, they can easily introduce cycles in a search path. To alleviate the problem, search must maintain state to determine which non-terminating rules it has already applied and to which trees. In contrast, GIST performs a transformation step before the search to rewrite source and target instruction patterns into a canonical form, reducing, but not eliminating, the need to apply axioms during search. (We discuss that in Chapter 6.)

Axioms, used as rewrite rules to transform a tree into a normalized form, constitute a *term rewriting system* (TRS) [Baader and Nipkow, 1999]. In a TRS, one can reach a normal or canonical form for every expression only if the rule set is terminating and confluent. Although most of the axioms we apply to instruction patterns are easy to verify by inspec-

tion, we must still handle the important cases of commutativity and associativity. For this reason, GIST's canonicalization process occurs in two steps: first, normalize instruction patterns using a subset of rules that are terminating and confluent; second, use *expression reassociation* to handle those algebraic properties that are potentially non-terminating.

5.2 Axiom Normalization

A central component of GIST is to syntactically match semantic definitions of IR to target instructions to determine equivalence. We compare each subtree of an IR pattern to the set of target instructions to find an implementation of that IR on the target. If a match fails on a subtree, we conclude that an implementation of that subtree on the target does not exist. The ability to write CISL descriptions in many different, yet semantically equivalent, ways has a significant impact in our ability to perform this syntactic matching process. For example, consider the following IR and target CISL statements:

IR	Target
`if (!(field > 0)) { ... }`	`if (field <= 0) { ... }`

The IR semantics describe a conditional that tests whether the complement of **`field`** is greater than **`0`**, whereas the target semantics describe a conditional that tests if **`field`** is less than or equal to **`0`**. Both semantics mean the same thing, yet a straightforward matching procedure would discover that they are syntactically different, thus concluding that they represent different semantics.

Differences can also arise when the target machine does not support the semantics of an IR instruction directly. For example, imagine we are trying to match an IR instruction, performing a basic subtraction operation, against a minimalistic instruction set such as the PDP-8. Unfortunately, the PDP-8 does not provide a subtraction instruction making it

impossible for a match to be found—yet it is certainly possible to represent subtraction in terms of addition and negation[1] as shown in the following identity:

$$X - Y \quad \approx \quad X + (- Y)$$

This identity is used as a *rewrite rule* to translate a subtraction operation on two subtrees, X and Y, into an addition of X to the negation of Y. Using identities as rewrite rules allows IR pattern trees to be matched and rewritten into different, yet semantically equivalent, forms—allowing target instructions to be matched that would have otherwise not been possible.

Prior work emphasized the use of rewrite rules during the search process. For example, in Cattell's thesis work [Cattell, 1978] he applied semantics preserving axioms when the search process was "stuck" (i.e., a matching target instruction could not be found). In this approach, axioms were used in a bi-directional manner: both the left- and right-hand side of a rule could be considered to rewrite an IR subtree. Although axioms define legal moves in the search space, and search finds the correct combination of axiom applications, the process has the potential to be combinatorially explosive. Consider a rule defining commutativity of addition (a legal axiom in Cattell's work):

$$X + Y \quad \leftrightarrow \quad Y + X$$

This rule is non-terminating, i.e., it can be applied ad infinitum, producing cycles in the search path. To counteract this effect, search must maintain state to recognize which axioms have been applied, the application frequency, and/or the depth of the search tree, to prune potentially problematic paths. Although Cattell found this approach to work in most cases, he was dealing with relatively simple semantic trees. In particular, his target instruction semantics are written in terms of the IR, had identical addressing modes, and used only one or two operators in a given tree. In this work, we are dealing with addressing modes

[1] In fact, the PDP-8 does not even have a negate instruction requiring additional translation leveraging rules relying on the underlying properties of two's complement arithmetic.

that may not be represented exactly on the target, and IR trees involving complicated semantics (e.g., JVM bytecode). Indeed, in a preliminary investigation we attempted Cattell's approach, which led to long search times that often resulted in a failed search.

To fix this problem and benefit from semantics preserving axioms, GIST applies *axiom normalization*. Axiom normalization leverages techniques from term rewriting systems to reduce semantic trees to a canonical form. This process departs from earlier work in that the normalization procedure is applied *before* search and to both IR and target semantic trees. Thus, the objective is to use rewrite rules to impose a standard shape on semantic trees as a preprocessing step before search to eliminate differences in specification, thus improving the matching process. To show how this works, we describe CISL semantics as terms (a mathematical structure used by TRS) and describe the necessary properties required by our term rewriting system to produce normal forms: *termination*, *confluence*, and *admissibility*.

5.2.1 Terms and Equations

The fundamental structure used by a term rewriting system is a *term*. Formally, consider a non-empty set of variables X and a set of operators Σ where each operator $f \in \Sigma$ has an arity n denoted by f^n. Each operator of arity i is contained in the set $\Sigma^{(i)}$ and the operators with arity 0, called *constants*, are denoted by $\Sigma^{(0)}$. A *term* is defined as follows:

1. A variable $v \in X$ is a term
2. A constant $c \in \Sigma^{(0)}$ is a term
3. If $t_1 \ldots t_n$ are terms and $f \in \Sigma^{(n)}$, then $f(t_1, \ldots, t_n)$ is also a term.

Given this definition, it is clear that we can represent CISL semantics as terms. CISL binary and unary operators belong to the sets $\Sigma^{(2)}$ and $\Sigma^{(1)}$ respectively. Memory indexing can be viewed as a binary operation s that takes a memory m and an index i and evaluates to the location's value: i.e., the CISL code `M[x]` is identical to the term $s(M,x)$. In addition,

integer and floating-point literals and field names are represented by term constants. Lastly, variables extend the CISL syntax to allow arbitrary subtrees to be referred to in rewrite rules.

Now let $T(X,\Sigma)$ be the set of all possible terms, and let t_i and t_j be elements of $T(X,\Sigma)$. We can define a set of equations E over T such that t_i is congruent to t_j if they can be made syntactically identical through equational rewriting and variable substitutions using equations in E. In this work, $C = T(X,\Sigma)$ are those terms that represent valid CISL semantic definitions and E is a set of equations that define semantics preserving axioms. To make this clear, consider again the following rule for subtration:

$$\text{X - Y} \quad \approx \quad \text{X + (- Y)}$$

In this example, X and Y are variables and + and - are operators in $\Sigma^{(2)}$ and $\Sigma^{(1)}$ written in infix notation. By our definitions, this rule is an equation in E over the terms defined for CISL extended with variables. Now assume that t_{IR} = `4 - 5` and t_{Target} = `4 + (-5)`, where t_{IR} is a subtree in a CISL definition for an IR instruction and t_{Target} is a subtree found in a target instruction. Then, t_{IR} is congruent to t_{Target} precisely when X is 4 and Y is 5. Showing that two terms are congruent is identical to determining if two CISL subtrees are equivalent. Thus, given the set of terms defining CISL semantics, C, and a set of equations $E(C)$ that define semantic equivalence in C, and the equality predicate $=$, is $t_{IR} = t_{Target}$ *true* in our equational model $E(C)$? This problem is often referred to as the *word problem* in an equational theory, and term rewriting systems provide a framework for determing its solution.

5.2.2 Term Rewriting Rule of Inference

To determine if two semantic definitions are congruent we require a reasoning procedure that will allow us to infer equivalent semantics. A term rewriting system provides a single inference process that allows us to accomplish this task. A term rewriting system R is a set of finite rewrite rules where each rewrite rule is a directed equality of the form $l \rightarrow r$ where l and r are terms in $T(X,\Sigma)$ and $l > r$ according to some defined ordering on terms.

A rewrite rule $l \rightarrow r$ is applicable to some term $t \in T(X,\Sigma)$ if there is an occurrence of a subterm l in t. The process of applying a rewrite rule x to t is called a reduction on t using x.

This process can be more formally stated by the following: let $t(e)$ be the term we are trying to rewrite where e is a distinguished subterm of t. The *term rewriting rule of inference* is:

$$\frac{\begin{array}{l} t(e) \\ l \rightarrow r \end{array}}{t(\sigma r)}$$

where σ *is a most general substitution such that* $\sigma l \equiv e$

That is, the left-hand side l of the rule is matched against the subterm e and this subterm is replaced by the right-hand side r. This process of matching e and l is a restricted form of unification known as *one-way matching* (as only one term contains variables) and the substitution σ is called a unifier of e and l. In other words, the unifier is a mapping of the variables in l to terms in e. This mapping is then applied to the right-hand term r of the rule to produce σr.

If no such mapping σ exists for any $x \in R$ applied to a term t, the term is said to be *irreducible* in R. That is, t is irreducible if there are no rules in R that can reduce t. The notation $t_i \stackrel{R}{\Longrightarrow} t_j$ denotes that a term t_i is reducible to t_j by one or more rewrite rule applications in a term rewriting system R. To illustrate this deduction mechanism, consider the following set of rewrite rules *Nat*,

$$\begin{array}{lrcl} \text{(R1)} & x + 0 & \rightarrow & x \\ \text{(R2)} & x + \text{s}(y) & \rightarrow & \text{s}(x + y) \end{array}$$

These rules axiomatize addition on the natural numbers where 0 is a constant in $\Sigma^{(0)}$, x and y are variables in X, s and $+$ are operations in Σ having arity 1 and 2 respectively, and for all

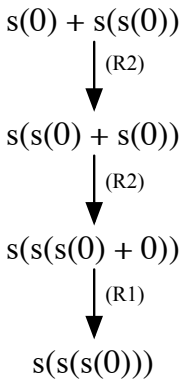


Figure 5.1: Reduction of the term s(0) + s(s(0)) in *Nat*.

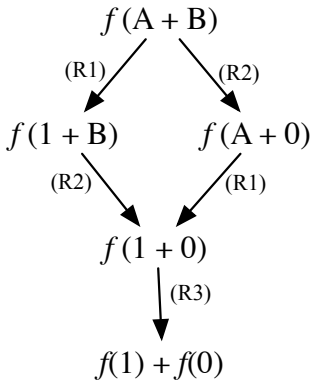


Figure 5.2: Reduction of the term f(A + B) in *Con*.

the rules $l \rightarrow r$, l and r are terms in $T(X,\Sigma)$. Using these definitions, we can represent the natural numbers 1 and 2 as the terms s(0) and s(s(0)) and the addition of 1 and 2 as the term t_1 = s(0)+s(s(0)). Using rules (R1) and (R2) and the rewriting rule of inference described above, we show the reduction of $t_1 \in$ *Nat* in Figure 5.1. This form of reduction characterizes a form of computation. That is, through successive rule applications in *Nat*, we deduce that s(0)+s(s(0)) $\stackrel{Nat}{\Longrightarrow}$ s(s(s(0))) or, in a more conventional notation, that 1+2 $\stackrel{Nat}{\Longrightarrow}$ 3. Thus, the rules of *Nat* axiomatize the natural numbers (represented by the *successor* function *s*) and addition. The reduction process emulates addition producing a term that represents the result of adding the numbers 1 and 2. The resulting number 3 can be considered a normal form of the expression 1 + 2 when expressed appropriately.

Although the rules for *Nat* always terminate with a unique normal form, is this the case for all term rewriting systems? That is, does the application of rules in a term rewriting system always terminate and consistently produce the same reduced term? Consider the following set of rules, *Con*:

$$\begin{array}{lrcl} \text{(R1)} & \text{A} & \rightarrow & 1 \\ \text{(R2)} & \text{B} & \rightarrow & 0 \\ \text{(R3)} & f(x+y) & \rightarrow & f(x)+f(y) \end{array}$$

Given the term $f(\text{A} + \text{B})$ we apply the rules (R1)-(R3) to show in Figure 5.2 that two reduction paths are possible and each path terminates with the same normal form. We use this example to illustrate two fundamental properties of term rewriting systems:

Termination Is it always possible, given a set of rules, that the process of applying those rules will terminate? For the set of rules in *Con* this is certainly the case. What happens if we include the following commutivity-of-addition rule in *Con*:

$$\text{(R4)} \quad x+y \rightarrow y+x$$

We now have a situation where (R4) can be applied infinitely many times and the reduction process will never terminate. Furthermore, it is easy to imagine a case where non-termination is the result of a subset of rules that work together to produce an infinite chain of reductions.

Confluence Figure 5.2 demonstrates the possibility for different reduction paths to lead to a common term. More specifically, given the term t it was possible to derive two different terms t_1 and t_2 and those terms could be *joined* to derive the term s. This property of term rewriting systems is known as *confluence*. The question then is, is it possible for these reduction paths to lead to a situation where t_1 and t_2 cannot be joined? More formally, given some term t and the reductions $t \overset{R}{\Longrightarrow} t_1$ and $t \overset{R}{\Longrightarrow} t_2$

for some term rewriting system R, is it possible that $t_1 \overset{R}{\Longrightarrow} s$ and $t_2 \overset{R}{\Longrightarrow} s'$ such that $s \neq s'$? While *Con* is certainly confluent, consider the addition of the following rule:

(R5) $x + 0 \rightarrow x$

This leads to two different reductions that result in different terms that cannot be joined, as depicted in Figure 5.3.

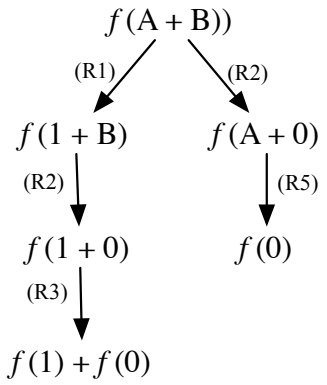


Figure 5.3: $f(1) + f(0)$ and $f(0)$ cannot be joined.

If a term rewriting system R is both terminating and confluent then R is said to be a *canonical term rewriting system*. This is not only interesting from a theoretical standpoint but also has practical implications. In this work it is critical that the application of rules that define equivalence of instruction semantics terminate and that the terms resulting from a chain of reductions lead to a unique canonical form. This means that CISL semantics that are expressed differently, but mean the same thing, will be reduced to the same normal form given a set of semantic preserving rewrite rules that constitute a canonical term rewriting system. Using a well known procedure, known as *completion*, allows us to determine if a given TRS is terminating and confluent. If the rule set is not confluent, the completion procedure will try to generate a new set of confluent rules or fail. If the rule set is non-terminating, the procedure does not terminate [Baader and Nipkow, 1999]. Fortunately, the axioms we use to canonicalize CISL pattern trees are simple and can be verified easily by inspection (see Figure 5.4). Although termination and confluence are restrictive in the kind

of rules we may include, it allows normal forms to be guaranteed. As it turns out, for our purposes, we must further restrict an axiom to be *admissible* so we do not exclude valid target sequences during search—we discuss axiom admissibility in the next section.

5.2.3 Axiom Admissibility

Axioms can be used to describe a wide range of semantic equivalences. In the case of canonicalization, we must take care to define rules that do not violate a canonical term rewriting system or exclude potential target sequences. We call an axiom that satisfies these conditions an *admissible axiom*. Although a complete TRS can be guaranteed using a completion procedure, our current rule set for canonicalization is straightforward and can be verified by inspection.

To ensure an axiom does not exclude target sequences we verify that the following properties do not exist in its definition: *constant introduction*, *constant elimination*, and *operation elimination by constant*. Consider the following axiom that introduces a constant on the right-hand side of the rule:

$$\mathrm{R[X]} \quad \rightarrow \quad \mathrm{R[X] + 0}$$

This rule transforms all R memory references into equivalent references adding 0. The first problem we notice is that the axiom is non-terminating. If we ignore this issue, however, we observe that by introducing the constant 0 on the right-hand side we are unable to find target patterns that reference just the memory R. In the case of the MIPS architecture, this means we could not match the SW (store word) instruction directly; rather we would first require an ADDI (add immediate) to produce a result into a sole register and could then perform the store. This excludes the more efficient (and sensible) sequence of just SW, thus the axiom is inadmissible. We discuss more details of the matching process in Section 6.5.2. Alternatively, consider the elimination of a constant:

$$\mathrm{R[X] + 0} \quad \rightarrow \quad \mathrm{R[X]}$$

In this case, the rule transforms the addition of the memory reference R by 0 into just the reference to R. This would exclude the possibility of finding a target sequence requiring the addition operation. This is exactly the case when dealing with minimal instruction sets such as the PDP-8. In Section 6.5 we provide an example that demonstrates exactly this case causing a target sequence to not be found. By the same argument, the following axiom eliminating the multiplication operation by a constant is also inadmissible:

$$X \times 0 \quad \rightarrow \quad 0$$

This axiom reduces a multiplication by 0 to just the constant 0. This unnecessarily restricts the number of target sequences we could find on the target.

In Figure 5.4 we show examples of admissible axioms used by canonicalization. Most of these rules do not contain constants and others clearly satisfy the conditions for admissibility. The majority of these axioms emphasize algebraic properties for boolean and relational operations and only provide a few axioms for arithmetic. This makes sense as most arithmetic laws are either unacceptable for a complete TRS (e.g., commutativity, associativity) or the additional conditions we impose for admissibility. It is important to emphasize that the rules we chose for canonicalization are motivated by the need to generate normalized forms for both the IR and ISA pattern trees. In particular, they focus on simplifying trees in a way that enables expression reassociation (discussed in Section 5.3) to reorder subtrees using laws of commutativity and associativity. For example, a CISL expression of the form X-Y+Z can be rewritten to X+(-Y)+Z using the canonicalization axiom X-Y $\rightarrow$ X+(-Y) allowing it to be reordered by commutativity of addition. Thus, our admissible axioms are specifically chosen to work in conjunction with our expression reassociation procedure to improve the overall effectiveness of the normalization process. The use of inadmissible axioms is still important for finding target sequences—they are used to rewrite IR trees during heuristic search. However, the effectiveness of canonicalization reduces their importance.

Boolean Axioms	**Arithmetic Axioms**	**Relational Axioms**	**Bitwise Axioms**
!(!X) → X	-(-X) → X	X == Y || X > Y → X ≥ Y	~(~X) → X
!(X && !Y) → !X || Y	-0 → 0	X == Y || X < Y → X ≤ Y	~(X & ~Y) → ~X | Y
!(X || !Y) → !X && Y	X - Y → X + (-Y)	!(X == Y) → X ≠ Y	~(X | ~Y) → ~X & Y
!(X || Y) → !X && !Y		!(X > Y) → X ≤ Y	~(X | Y) → ~X & ~Y
!(X && Y) → !X || !Y		!(X < Y) → X ≥ Y	~(X & Y) → ~X | ~Y
!(!X || !Y) → X && Y		!(X ≤ Y) → X > Y	~(~X & ~Y) → X | Y
!(!X && !Y) → X || Y		!(X ≥ Y) → X < Y	~(~X | ~Y) → X & Y
X && X → X			X | X → X

Figure 5.4: Admissible Axioms

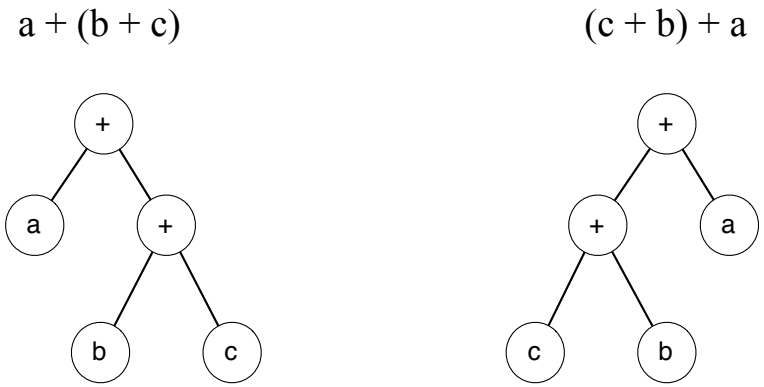


Figure 5.5: Commutativity/Associativity Example

5.3 Expression Reassociation

The goal of GIST is to find target sequences that are semantically equivalent to a compiler IR instruction. This is done by matching the structure of an IR pattern tree to target pattern trees. Trees that are structurally the same are semantically equivalent. This is problematic when a binary operation is commutative or associative. Consider the two expression trees shown in Figure 5.5. By the laws of commutativity and associativity of addition these two arithmetic expressions are equivalent. Below each written expression is its structural representation as a tree. Comparing these trees node by node would indicate that they are not the same. We use expression reassociation, a technique based on global reassociation used to improve partial redundancy elimination, to order the child nodes of commutative/associative operations to produce trees that are structurally identical.

As mentioned previously, CISL semantic descriptions are represented internally as trees. Each node in the tree is associated with a *kind* that identifies a statement or operation (e.g., sequence, if, +, -). Each node is also assigned a *rank*. To re-orient tree nodes we follow the following steps:

1. Compute a rank for every node in the tree
2. Reassociate expressions by sorting their operands
3. Identify *compiler constants*

The next three sections describe these steps in more detail. To help understand the procedure we provide a running example illustrating each step. In Figure 5.6 we show the CISL code for the Baseline **`iadd`** instruction (as discussed in Section 3.3) and its translation into the internal GIST pattern tree.

5.3.1 Computing Ranks

To guide reassociation, our first step is to rank each tree and subtree. Given a GIST pattern tree we visit the nodes bottom-up assigning ranks using the following rules:

```
S.slot[direct(spTopOffset + 4)] =
       S.slot[direct(spTopOffset)] +
       S.slot[direct(spTopOffset + 4)];
```

CISL Syntax

```
(lset (lget S slot (+ spTopOffset 4))
      (+ (lget S slot spTopOffset)
         (lget S slot (+ spTopOffset 4))))
```

GIST Pattern Tree

Figure 5.6: CISL Baseline iadd Instruction

1. Constants receive rank 0.
2. Instruction parameters receive rank 1.
3. Memory references receive rank 2.
4. An expression tree receives a rank equal to its highest ranked operand.
5. All other nodes are statements and do not receive a ranking.

In Figure 5.7 we show the rank assignment for our example pattern tree. All appearances of the instruction parameter **`spTopOffset`** get rank 1, the constant **`4`** gets rank 0, the **`+`** operation used to index the stack, **`S`**, has rank 1 (equal to its highest ranked operand **`spTopOffset`**), the **`lget`** memory references receive rank 2, and the final **`+`** performing the actual addition gets rank 2.

5.3.2 Operand Sorting

Given the ranked trees, we are almost in a position to reassociate expressions. Unlike global reassociation, we are not required to rewrite expressions of the form $z-y+x$ into $x+(-y)+z$ because our axiom normalization process takes care of this. We do rewrite binary trees for associative operations of the form **`(+ x (+ y z))`** into their n-ary equivalent **`(+ x y z)`** to eliminate any ambiguity. We then recursively visit subtrees, sorting the

```
(lset (lget S slot (+ spTopOffset:1 4:0):1):2
      (+ (lget S slot spTopOffset:1):2
         (lget S slot (+ spTopOffset:1 4:0):1):2):2)
```

After Rank Assignment

```
(lset (lget S slot (+ 4 spTopOffset))
      (+ (lget S slot spTopOffset)
         (lget S slot (+ 4 spTopOffset))))
```

After Reassociation

```
(lset (lget S slot (cconst (+ 4 spTopOffset)))
      (+ (lget S slot spTopOffset)
         (lget S slot (cconst (+ 4 spTopOffset)))))
```

After Compiler Constant Identification

Figure 5.7: CISL Reassociation Transformations

operands of commutative operations from low to highest. Lower ranked operands (e.g., constants, parameters, memory references) are grouped together followed by expression trees.

Figure 5.7 emphasizes in bold the result of sorting trees by rank. Here we notice that in two cases the constant **4** and parameter **spTopOffset** are reoriented. Because the ordering and grouping of operands and subtrees are known we can leverage this information to identify *compiler constants* (discussed in the next section). Other important results of reassociation include instructions operating on immediate values, such as a register add immediate instruction **R[x] + imm**. For this case reassociation guarantees that this will always be rewritten into **imm + R[x]**. This allows semantic trees to be indexed according to their shape making it easier to choose likely target instructions quickly. We describe the indexing scheme in Chapter 6.

5.3.3 Compiler Constants

When generating code for a target machine it is important to identify constant expressions at compile-time. This allows the expression to be computed by the compiler rather than

generating code to perform the operations at run-time. Likewise, it is important for GIST to identify constant expressions found in an instruction's semantic description. This allows search to treat an entire expression as a constant leading to a reduction in the number of target instructions required to implement an IR. In our example in Figure 5.7 we see that two expressions of the form **`(+ 4 spTopOffset)`** were identified as compiler constants. Identifying compiler constants in a reassociated pattern tree is easy because constants will always appear first and instruction fields second in an operators parameter list. Thus, a compiler constant is created by examining the parameters left to right. If the parameter is a constant or field, we include it as part of a compiler constant. The compiler constant is constructed when we encounter a non constant or field subtree.

Transforming these expressions into compiler constants allows target sequences to be found that do not require two additional add immediate instructions. Indeed, without compiler constants the best PowerPC sequence GIST could find was an add, followed by two memory loads, another add, and a store (it only required a single add for the constant expression because we identify common sub-expressions during search time). With compiler constants we were able to find the optimal sequence of two loads, an add, and a store. Information about constant expressions is propagated into the instruction selector patterns generated by GIST. We can take advantage of this information in a compiler adapter (discussed in Chapter 7) to generate compiler code to compute the expression at compile time.

5.3.4 Canonicalization Example

Consider the following description of the JVM **`iadd`** semantic pattern for Baseline:

```
Stack[spTopOffset+4+FP] = Stack[spTopOffset+FP] +
                          Stack[spTopOffset+4+FP];
```

The JVM specification [Lindholm and Yellin, 1999] describes the effect of this instruction as popping the top two values off the operand stack, adding those values, and pushing the result. Baseline modifies these semantics by introducing a new argument, **`spTopOffset`**, to the instruction that Baseline manages at compile time (as mentioned previously). We

show the canonical form below using a store mapping for PowerPC (**Stack** ⇒ **M** and **FP** ⇒ **R[1]**). We represent the simplified view of the canonical form in a LISP-like tree format:

```
(if true
  (lset (lget M (+ (ct-const (+ 4 spTopOffset))
                   (lget R 1)))
        (+ (lget M (+ spTopOffset (lget R 1)))
           (lget M (+ (ct-const (+ 4 spTopOffset))
                      (lget R 1))))))
```

Because the **iadd** instruction always executes, its guard is the value **true**. The **lset** and **lget** operations indicate a store and load operation to a memory location respectively. (If an **lget** occurs as the first argument to an **lset** it is used as a location rather than as a value; this allows easier common subexpression matching for address computations.) Using expression reassociation allows us to identify constants that we can compute at code generation time. We identify these compile-time constants with the **ct-const** annotation. The actual canonical form includes additional information, such as types for all the expressions in the pattern. We drop this information from the example to make it readable.

5.4 Summary

In this chapter we described GIST's canonicalization procedure, a novel approach used to improve semantic matching. To allow syntactically different, but semantically equivalent, CISL trees to be matched, GIST applies a two-phase normalization process. The first phase uses a term rewriting system composed of semantic preserving axioms that are complete and admissible to reduce trees to a normal form. A term rewriting system that is complete, terminates after a finite number of rule applications and always reduces trees to a normal form (confluent). An admissible axiom ensures that target instructions are not excluded by the rewriting process. The second phase uses expression reassociation to re-orient subtrees using a well defined ranking scheme. This handles problematic (e.g., non-terminating) axioms such as commutativity and associativity of addition. Through the reassociation

process we identify compiler constants that improve target sequences by identifying expressions that can be computed at compile-time.

CHAPTER 6

DISCOVERING INSTRUCTION SELECTOR PATTERNS

6.1 Overview

The instruction selector (IS) phase of the compiler translates each IR instruction, generated by the front-end, into a sequence of target instructions. Choosing the right target sequence for each IR instruction is dependent on several factors, including the IR operation, the instruction operands (e.g., temporaries, immediates), constraints on those operands, and compiler implementation decisions. Although each IS design is different, they all describe IR patterns for matching IR operations and a corresponding target template for generating a target sequence. Together they form an instruction selector pattern matching rule. These IS pattern matching rules can be implicit in the form of implementation code or explicit as a set of rule descriptions. For example, a typical hand-written IS encodes these patterns as a sequence of if-then-else statements that select the appropriate target instructions by examining IR data structures (as we described in Section 2.2). A BURS-generated IS specifies patterns as trees that are used to generate tables to drive the instruction selection process (see Section 2.3). Although each of these examples (and others outlined in Chapter 2) solve the IS problem differently (and manually), they are similiar in that they all use patterns to describe the translation from IR to target sequences.

The input to GIST's search procedure is a set of source and target instruction patterns described in CISL (we refer to target templates as patterns since they are described in the same language.) To keep the discussion simple, we henceforth refer to these as a compiler IR and an ISA. The goal is to find, for each IR pattern, a sequence of ISA patterns that im-

plements the semantics of the IR operation on the target architecture. Recently, Dias [2008] proved that finding a mapping from IR to target sequences is an undecidable problem. For this reason, the application of heuristic search is necessary for generating instruction selector patterns automatically. Related work (e.g., [Cattell, 1978]) has mainly employed depth-first search on the IR tree, attempting to match target patterns to the *root*. Thus, it builds a target sequence from the last-executed instruction back to the first. In addition to matching the root operator exactly, top-down matching may explore transforming the IR tree to use a *related* operator. For example, if a target has no subtraction instruction but does have unary negation, then rewriting $a-b$ to $a+(-b)$ will obtain a match, while $a-b$ will not.

If a target pattern matches everywhere except its leaf nodes, top-down matching decomposes the problem and attempts to find a separate match for the subtree rooted as the mismatching leaf, and that leaves its result in a place that matches the target pattern's leaf (most frequently a register). This approach has been used with some success in the past, but it spends significant time dealing with mismatches. In addition, because it finds target instructions in reverse execution order it is harder to reuse equivalent locations (i.e., a register that has already been loaded with an equivalent value in memory [Ganapathi et al., 1982]) and common subexpressions more generally.

GIST departs from prior approaches by matching the IR tree *bottom-up* and *breadth-first*. It attempts to match each unique subtree of the IR pattern at each search point against target patterns. Whenever it finds a match, it *reduces* the IR subtree to the destination location of the target pattern. For example, given the following tree pattern:

```
(if true (lset (lget M (+ (lget R 4) imm))
               (+ (lget R 5) (lget R 6))))
```

it would simultaneously try to match and reduce the complete tree, and also the left-hand side subtrees under **`lset`**, namely:

```
(lget M (+ (lget R 4) imm))
(+ (lget R 4) imm)
```

```
(lget R 4)
imm
```

and the right-hand side subtrees:

```
(+ (lget R 5) (lget R 6))
(lget R 5)
(lget R 6)
```

Leaf nodes such as **`(lget R 4)`** and **`imm`** might match target data movement instructions, while larger expression trees might correspond to arithmetic operations on the target. This is quite similar to how BURS matches IR trees bottom-up to identify the correct predetermined target mapping. However, we use it here to find those mappings *automatically*, and to find *all mappings*, not just the locally cheapest one. The generated successor search states include the entire IR tree with the matched subtree replaced by the target location. Search discards successor states that do not meet a given efficiency metric compared with partial and full matches it has already found. The general search technique used is a bounded breadth-first search called *beam search* [Russell and Norvig, 2003].

Because it matches IR trees bottom-up, GIST finds target sequences in forward execution order, allowing us to leverage locations that hold identical values and apply forward analysis optimizations such as common subexpression elimination during search, leading to an overall reduction in search time and better target sequences. Our search approach is particularly advantageous for register-sensitive machines such as the IA-32. Furthermore, because we consider all possible subtrees, we can find all possible execution orderings of the target patterns (subject to the width of the beam search). This allows us to identify more efficient sequences and handle machines that have heavily constrained memory references.

Before we show an example of search and matching, we describe each component of GIST in detail. Figure 6.1 shows a data flow diagram of each component, inputs, and output of the GIST architecture. In previous chapters we have discussed the CISL language, store mapping and schemas, canonicalization, admissible axioms, expression reassociation, and term rewriting systems. In this chapter we focus on the rest of GIST, including

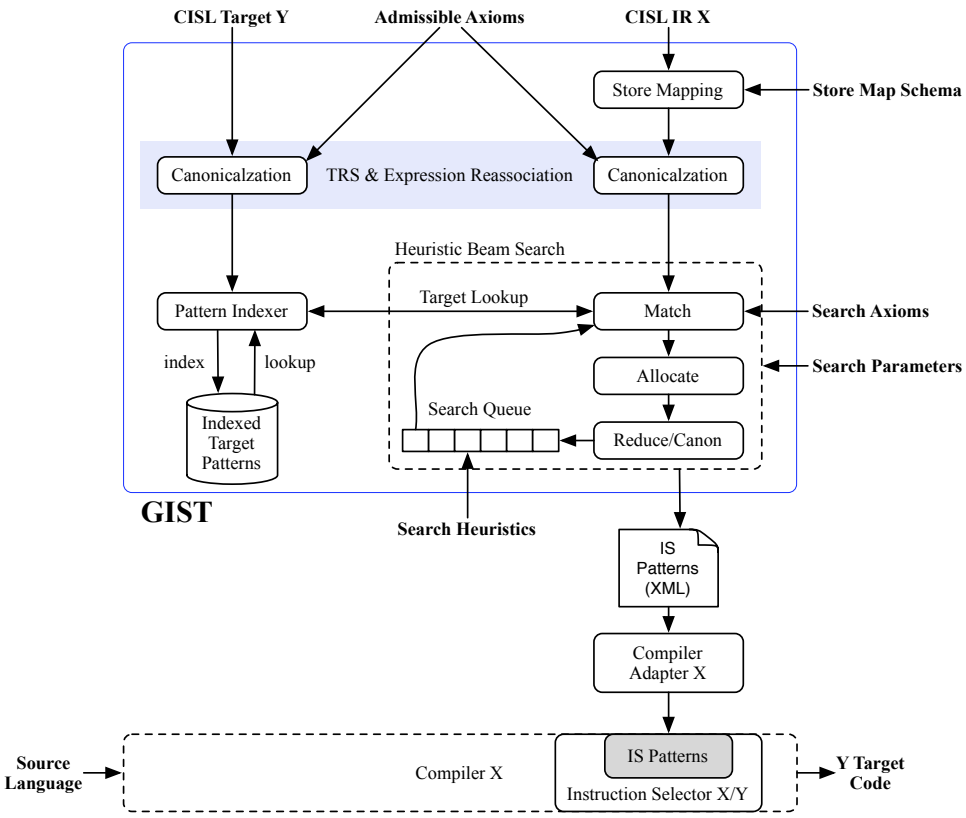


Figure 6.1: GIST Architecture

fill me in

Figure 6.2: Overview of Search and Match Process

how GIST finds candidate target patterns quickly, how it allocates resources (e.g., scratch registers) when a match succeeds, the matching process, semantics preserving axioms, the beam search procedure, and search-related parameters and heuristics. First, we describe instruction selector patterns and their relationship to CISL descriptions.

6.2 Instruction Selector Patterns

Instruction selector patterns translate IR instructions to target sequences that perform the same operation (i.e., they are semantically equivalent). Each IS pattern is composed of an IR pattern and a sequence of target patterns: IR patterns *recognize* instances of IR in-

structions and target patterns *generate* instances of target instructions. The difference between an instruction pattern and instance is that the former specifies an instruction template whereas the latter corresponds to a specific instruction. These templates can be used for matching or generation purposes. Consider the following ISA pattern written in CISL:

```
R[rd] = R[rs] + imm;
```

In this example, an immediate (**imm**) and a register (**R[rs]**) are added together and a result is stored in a destination register **R[rd]**. The parameters **rs**, **imm**, and **rd** correspond to instruction variables defined elsewhere in the class definition of this instruction (see Chapter 3 for details). This example defines a pattern because it can refer to any number of possible add immediate instructions where **rd**, **rs**, and **imm** are instantiated with specific values. Now consider the following IR pattern:

```
T[s] = T[s] + 1;
```

This example is similar to the previous one. However, it defines an increment by one operation: a temporary register (**T[s]**) is incremented by the integer constant **1**. It is important to note that this is still a pattern as **T[s]** can refer to any temporary register from an infinite register set. It should be clear that we can use the IR and ISA pattern to form an instruction selector pattern. For example, consider the following two IS patterns:

i) `T[s]=T[s]+1` **=>** `R[rd]=R[rs]+imm` {**rd=s, rs=s, imm=1**}

ii) `T[s]=T[s]+i16` **=>** `R[rd]=R[rs]+imm` {**rd=s, rs=s, imm=imm16, fits(i16,8)**}

The left-hand side of **=>** in pattern i) *matches* an IR increment instance, where **s** is instantiated with a specific temporary register index, and the right-hand side generates the target instruction. A match is successful provided that the constraints, indicated between { and }, are satisfied. In this simple example, **rd** and **rs** must both refer to the same temporary index **s**, and **imm** must be the integer constant **1**. Pattern ii) is more similar to the target add immediate instruction. However, an additional constraint, **fits(i16,8)**, is introduced. If we imagine the **imm** ISA parameter to be 8 bits wide and the **i16** IR parameter to be 16

bits wide, this constraint ensures that an ISA instance can be generated only if **`i16`** *fits* in **`imm`**, that is, the bit width of the value of **`i16`** is less than or equal to the bit width of **`imm`** (e.g., 8 bits). The following table demonstates how these IS patterns match instances:

IR Instance	Match	Constraints	Target Instance
`T[4]=T[4]+1`	(i) *success*	`rd=4, rs=4, imm=1`	`R[4]=R[4]+1`
`T[4]=T[5]+1`	(i) *failed*	`s!=s'`	NA
`T[4]=T[4]+0xFF`	(ii) *success*	`rd=s, rs=s, fits(0xFF,8)`	`R[4]=R[4]+0xFF`

The first entry shows an IR instance that successfully matches IS pattern i) with all the constraints satisfied. This generates a target instance with each constraint bound to the corresponding target parameters. The second entry fails to match i) because the IR instance does not satisfy the constraints imposed by the IS pattern, namely 4 is not equal to 5. The third entry successfully matches pattern ii) as the bit width of the hexadecimal value **`0xFF`** *fits* in the target's immediate parameter **`imm`**. We are assuming that **`imm`** is unsigned—note that the match would fail if it is signed. Similar constraints are used to identify mismatches in signedness and endianness. GIST uses IR and ISA patterns defined in CISL descriptions to derive IS patterns automatically. Constraints are generated during the matching process and are used by search to determine the quality of a match and by an adapter to generate constraint checking code for the compiler's instruction selector. It is important to note that a successful match with constraints may be a more efficient pattern for a specific case, but it might not cover all possible IR instances of this sort. This raises questions about IS pattern completeness and coverage—thus, it is important for GIST to find not only the most *efficient* (e.g., special case) IS patterns, but also to find a set of IS patterns that collectively cover all IR instances. Proving that a set of IS patterns with constraints, having the lowest possible cost under each constraint, satisfies this condition would be useful for generating

the most efficient target sequences in all cases. We believe it is possible to encode this as a boolean formula suitable for input to a SAT solver. We leave this for future work.

6.3 Target Pattern Indexing

To accelerate matching, GIST indexes all target patterns according to the *shape* of their trees, i.e., taking into account the operators, but ignoring leaf variables and constants (which typically can unify with parameters of source trees). This is more general than prior work, which indexes only the root operators, and we gain considerable speed from it.

Because GIST considers all possible subtrees of an IR pattern, we extend primary operator indexing also to index data movement operations. This is important because we must consider matches on subtrees that are data locations (e.g., immediate fields, registers). For example, a target machine might not support an instruction for adding an immediate to a register value. However, it does support loading an immediate into a register followed by a register-register add. Thus, we would match the immediate field subtree on the IR to the immediate load instruction. This would reduce the immediate to the target register and allow a subsequent match to find the register-register add. Prior work considered loads and stores separately and applies different techniques to get data into the right locations before matching. GIST folds this into the same basic approach. Furthermore, because we canonicalize instruction patterns before search, we have considerable success without the need for axiomatic transformations.

In Figure 6.1, after the target CISL patterns are canonicalized, they are processed by the *pattern indexer* and stored in a table for efficient lookup. An ISA pattern is indexed in three primary ways to accommodate all possible IR subtree pattern configurations. In the simplest case, the entire ISA pattern is indexed. This corresponds to the case when an IR subtree matches an entire ISA pattern tree. Next, if an ISA pattern consists of multiple guarded effects (e.g., multiple CISL **`if`** statements), a separate index entry is included for each guarded effect. This recognizes the possibility that an IR subtree matches a partial

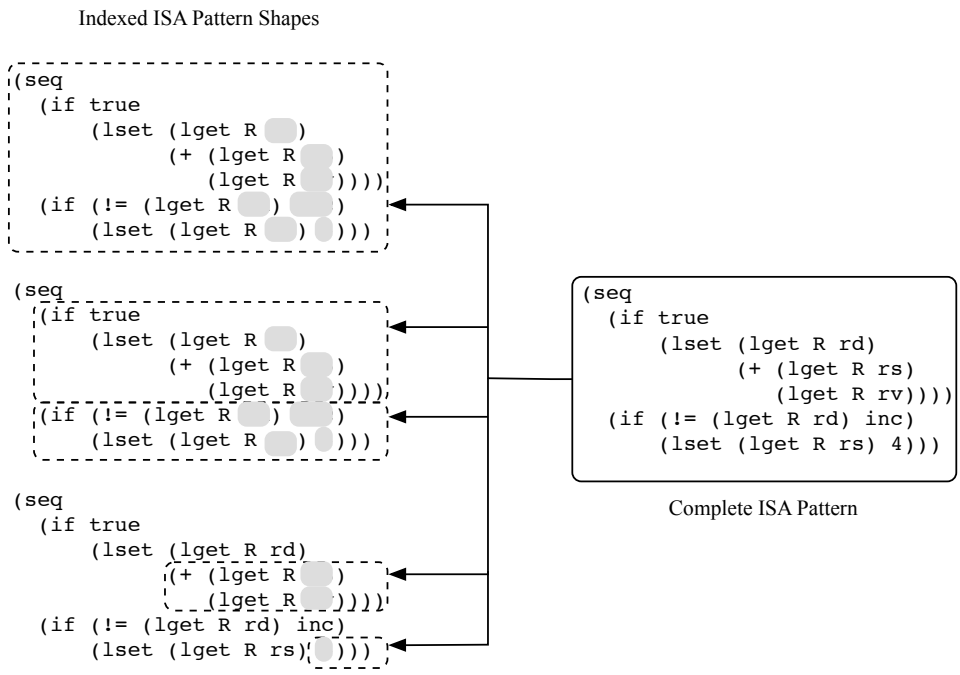


Figure 6.3: Target Indexing

ISA pattern. Lastly, a separate index entry is introduced for each right-hand side of an assignment (e.g., **`lset`**). This accounts for the case when an IR subtree matches only the effect of an ISA pattern. Side-effects and constraints are introduced for partial matches and are maintained by search to identify inconsistencies in partial solutions.

In Figure 6.3 we show the result of indexing an ISA pattern tree. The ISA pattern (on the right) is indexed in five different forms as shown by the indexed ISA pattern shapes on the left. In this example, an index entry, surrounded by a dotted box, represents each of the three ways an ISA pattern is indexed. Literal constants and parameters are hidden (gray ovals) as they are not considered by the indexing process—we are only interested in the shape of the tree. The top index entry corresponds to the entire ISA tree, the second and third entry are indexed by guarded effect, and the fourth and fifth entry are indexed by the right-hand side of an assignment.

At each search point, all subtrees of a partial IR pattern are used to look up indexed ISA patterns for that subtree. The candidate ISA patterns that are found, are then compared,

node-by-node, by a tree matching procedure. Consider the IR and indexed ISA patterns shown in Figure 6.4 (for simplicity we do not include all possible indexed entries). Each subtree of the IR is used to find a candidate ISA pattern. For example, consider all the IR subtrees:

```
(lset (lget R s) (+ 1 (lget R s)))
(+ 1 (lget R s))
1
(lget R s)
```

Each subtree is hashed by the pattern indexer to look up a candidate ISA pattern. For the first subtree, the pattern indexer would find a candidate match with ISA pattern 2, an add immediate instruction. The second subtree would find a candidate match with the right-hand side of the assignment in pattern 2. The third subtree matches the right-hand side of ISA pattern 1. In this case, pattern 1 is indexed as a data movement instruction (load immediate) on the **imm** parameter. It is important to note that we do not require special treatment for data movement operations—they are indexed like other instruction patterns. Prior work matched trees top-down, transforming trees when a mismatch was encountered—data movement is difficult to express with axioms, thus requiring special handling. The last subtree does not yield any candidate ISA patterns and would likely be discarded by search (unless a search axiom is used—this is discussed later in Section 6.5.3).

After one or more candidate ISA patterns are identified, a fine-grained matching process compares the IR subtree and the ISA indexed subtree to verify the match and potentially generate constraints (as mentioned earlier). If the IR subtree represents the entire IR pattern, the IR and ISA pattern are used to generate an IS pattern (with constraints). Otherwise, the IR subtree is reduced to the target location of the matched ISA pattern and search continues. For subtree **1** in the above example, the matching ISA pattern loaded an immediate into a register. The ISA **imm** parameter matches subtree **1** (the matching process is elaborated on later) reducing it to an allocated register. We discuss resource allocation in the next section.

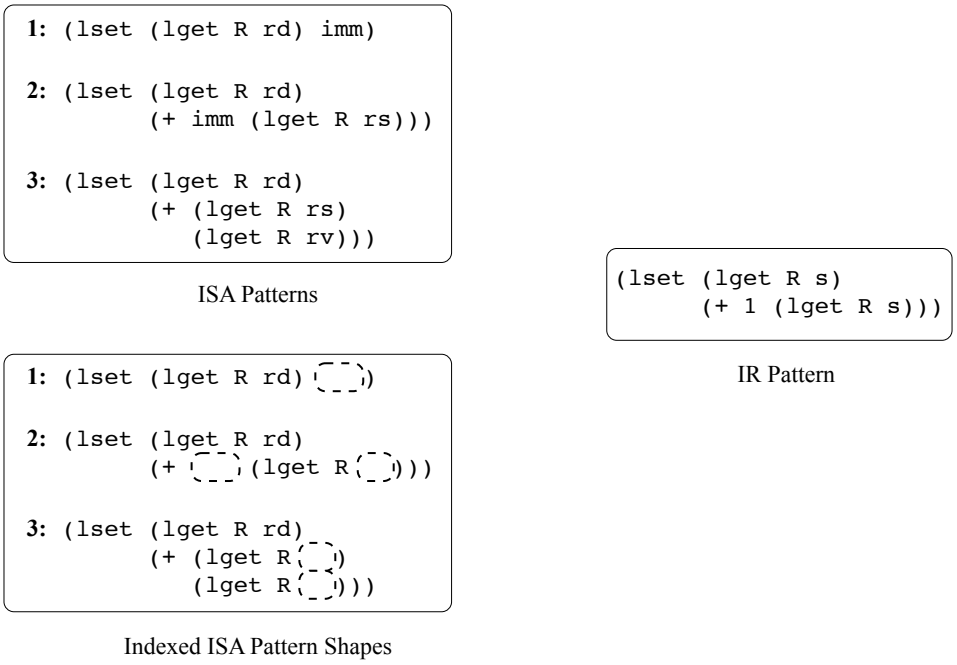


Figure 6.4: IR Lookup

6.4 Resource Allocation

To represent matching progress, when a target pattern matches an IR subtree, we reduce the IR subtree to the destination location of the target pattern. In many cases, that destination can be any one of a set of locations (i.e., general purpose registers). Further matching requires us to use a *particular location* rather than a pattern. Consider the PowerPC register-register add:

```
R[RT] = R[RA] + R[RB]
```

Assuming we find a match for the right-hand side addition, we reduce the IR tree to a specific assigned register, say **`R[5]`**, and not the pattern **`R[RT]`**. Reducing to the pattern would lose contextual information and prevent GIST from determining where a particular result is coming from when using it in later operations. In addition, it would be difficult to track locations containing identical values.

GIST assumes that actual resource allocation will occur in either a compiler adapter (described in Chapter 7) or in a later compilation phase (e.g., register allocation). Our

motivation for this is two-fold: universality and context. Some compilers may choose to perform register allocation in an earlier back-end phase, whereas others apply it as a final lowering step. Performing register allocation during search would preclude using GIST with many compilers and fail to meet our vision of generality. Because the kind of code generator patterns generated by GIST (and by most other code generators) are local (single IR instruction to a target sequence) it makes sense for register allocation to occur after generating longer target sequences, to take advantage of more global context.

For these reasons, GIST views the target machine as having an unbounded set of locations for each indexed store of the CISL description. When we reduce an IR subtree, we allocate a new scratch location from the target indexed store, i.e., one that does not conflict with any previously defined locations in the store. We track allocations that happen during a search and include them as part of each generated code generator pattern. Notice that if an IR pattern, perhaps via store mapping to the target, requires a *particular* element of the store, e.g., **`R[1]`**, the IR pattern will mention it explicitly so there will be no unbound pattern variables (such as **`RT`** in the PowerPC add pattern above). Rather, matching will bind **`RT`** to **`1`** in that case.

6.5 Heuristic Search

Finding an implementation of an IR instruction on the target is an undecidable problem. There are several factors that contribute toward this conclusion, including the difference in memory structures, the availability of ISA operations, the semantic complexity of the IR, and the variety of constraints imposed on particular operations. The level of difficulty increases when we generalize the problem to be compiler and target independent (i.e., we describe both IR and target machine). In particular, we can't make any assumptions about the structure of the compiler framework, its implementation, and the kind of instruction selector algorithms that it uses. This includes the compiler's definition of an efficient target

sequence. For example, are we generating code in a memory sensitive environment? Are shorter sequences important? Or is power a concern?

The difficult nature of the automatic instruction selector generation problem indicates that the solution requires heuristic search. In addition, we are not just looking for any solution (e.g., IS pattern), rather we are looking for the most efficient solution (typically referred to as an *optimization problem*). This provides guidance as to the kind of heuristic search technique we choose. Previous work focused on depth-first search, but spent a significant amount of time dealing with mismatches and backtracking. Another choice would be A* [Russell and Norvig, 2003] as it finds the least-cost path from an initial starting state (or node) to a goal node (out of several possible goal nodes). A* works best when an *admissible* heuristic function, h, can estimate the cost from some node x to a successor node y—an admissible heuristic never overestimates the actual minimal cost to reach the solution.

In our case, however, it is difficult to provide an admissible estimate because it is hard to determine which ISA instructions can be used next until we reduce a subtree. For example, matching an IR subtree to an ISA instruction may reduce that subtree to a specific location, which might allow the successor IR tree to be matched completely by a single ISA instruction in the next step. Unfortunately, this is entirely dependent on the ISA and thus difficult to generalize and to determine a priori. To do so, would require a search on the ISA just to generate an estimate. In effect, we would need to find a solution in order to generate a good heuristic estimate.

Alternatively, GIST uses a technique known as bounded best-first search, also called beam search. Beam search uses breadth-first search to build its search graph—at each search point it generates all possible successor states. Successor states are sorted in order of increasing heuristic values and a predetermined number of states are stored at each depth. We have found this approach capable of discovering IS patterns efficiently and quickly. The rest of this section provides an overview of GIST's version of beam search, the search

process, heuristics used to generate successor states, and the bottom-up matching process used to compare IR and ISA semantics.

6.5.1 Search Strategy

Search is depicted by a graph, where each node represents state that is maintained at each point during search. A node corresponds to either a partial solution to the problem or a *goal* node (an actual solution). In our case, a partial solution is a partially matched IR tree and a sequence of ISA patterns implementing the matched subtrees. Additional state is maintained for resources that have been allocated, constraints that have been generated, and axioms that have been applied. A valid move in the search graph occurs when an IR subtree can be implemented by an ISA instruction, thus expanding or generating a new node representing a new partial solution (reduced IR tree) or goal node. A path in the search graph is a sequence of IR-to-ISA matches that lead to a solution (if one exists). Figure 6.5 shows the structure and generation of search nodes in GIST. On the left, search begins with an initial IR pattern tree representing a complete IR instruction (IR_s). The initial state of this node is the CISL IR pattern tree. From IR_s three successor nodes, IR_1, IR_2, and IR_3, are generated, each holding information related to a partially matched solution. From each of these nodes additional successor nodes are generated until we reach the solution nodes, G_1 and G_2. We show part of the final state of G_1: an empty IR tree (fully matched), the matching target sequence (an add immediate instruction for some target ISA), and a set of constraints generated from the matching process.

Nodes are maintained in a single bounded priority queue (beam) ordered by decreasing heuristic value (discussed later). Thus, nodes toward the front of the queue represent partial solutions that are likely to lead to more efficient solutions. GIST uses a single queue (unlike traditional beam search) to compare partial solutions from a global perspective and to reduce the large amount of duplicate partial solutions that are generated (the queue removes identical partial solutions with identical heuristic cost values).

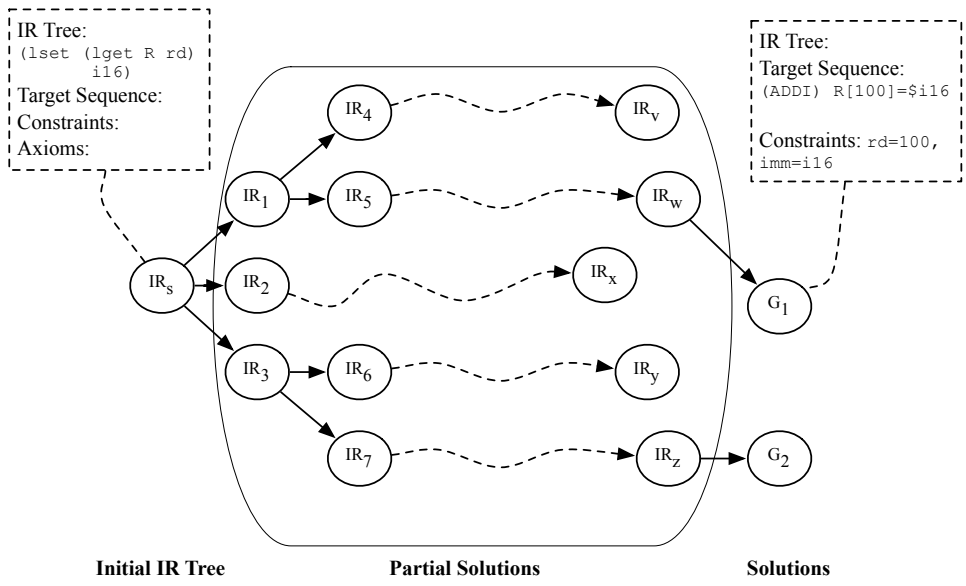


Figure 6.5: Search Graph

We choose a bounded queue because it reduces memory and time (number of nodes that need to be maintained and sorted) and the potentially infinite number of partial (but equivalent) solutions that can be generated at each search point. For example, a register-register add instruction is semantically equivalent to a sequence that stores the register values to memory, loads them back into registers, and then performs the addition. The store/load combination can be expanded infinitely, still resulting in a semantically equivalent, but less efficient, sequence. Because there will always be a more efficient version of these spurious sequences they typically reside at the back of the queue until they are pushed out by partial solutions move favored by the heuristics. Although this can exclude possible ISA sequences, we found that most sequences that are pruned exemplify this behavior.

6.5.1.1 Finding Candidate ISA Patterns

In Figure 6.6 we show a diagram of the GIST search process. First, search initializes the beam with a starting node (e.g., IR_s) containing a store-mapped and canonicalized CISL IR tree. The search algorithm then iterates, pulling nodes from the search queue, trying to find target patterns that match subtrees of a partially matched IR pattern. Termination is

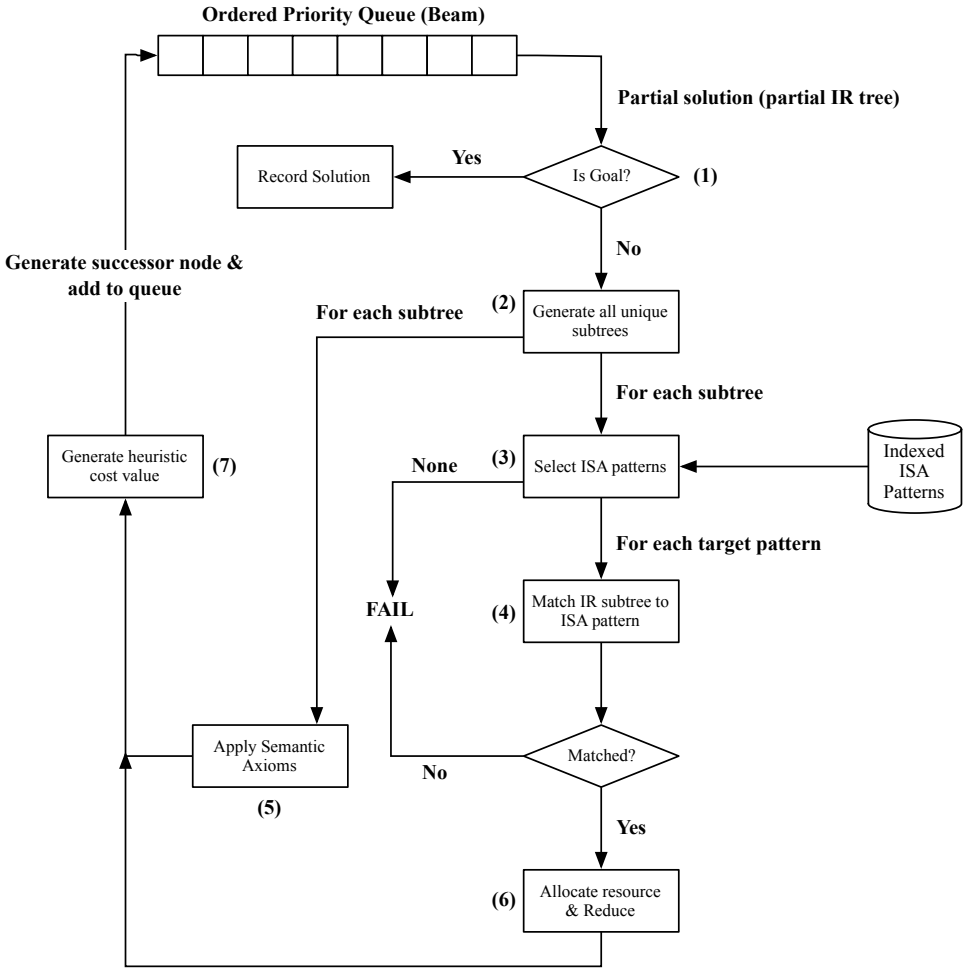


Figure 6.6: Search Flow Graph

based on a number of criteria including the number of solutions found, the number of nodes generated, the depth of the search graph, duration, etc. These parameters are provided by the user in an externally defined configuration file. However, most parameters assume reasonable defaults that work well with the compiler and target combinations we tested.

The first action performed by search is to check the partial IR tree to see if it is a goal node (step 1). A goal node is identified by a fully matched IR pattern, this is represented internally as an empty sequence tree: **`(seq)`**. If it is a goal node, an ISA pattern sequence has been found and the solution can be recorded. Otherwise GIST decomposes the partial IR tree into all possible subtrees (step 2). Because each subtree can be matched indepen-

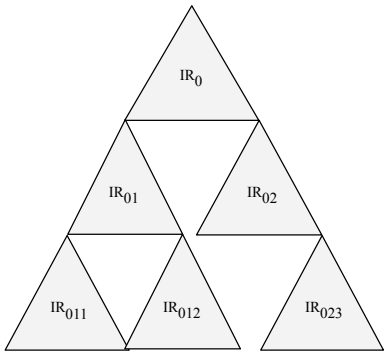


Figure 6.7: IR Subtrees

dently, the decomposition process must ensure that a matched subtree, and generated ISA instances, will not break semantic correctness of the IR. For example, an IR pattern that is a sequence of multiple guarded effects, $g_1, g_2, \ldots, g_n$ must find ISA patterns that match subtrees of g_1 before looking for an implementation for $g_2, \ldots, g_n$. This makes sense as each guarded effect represents an assignment that can change the state of the machine. Having these state changes occur in a different order can potentially result in a different machine state.

A CISL pattern tree is decomposed into several subtrees where each subtree represents an operator or a leaf (e.g., literal constant, compiler constant, instruction parameter). In Figure 6.7 we show an example of an IR tree that has been decomposed into subtrees. Each subtree is represented by a small gray-filled triangle labeled by its location in the tree. The top-most triangle, IR_0, represents the *entire tree* and has two subtrees, IR_{01} and IR_{02}. IR_{01} is a tree that also has two subtrees IR_{011} and IR_{012}, and IR_{02} is a tree that has a single subtree IR_{023}. At this point, the IR subtrees (IR_0, IR_{01}, IR_{02}, IR_{011}, IR_{012}, and IR_{023}) are transformed by semantics preserving axioms to generate new IR pattern trees (step 5 in Figure 6.6), and to look up candidate ISA patterns (step 3) that have been indexed by the pattern indexer (see Section 6.3). If we do not find any candidate ISA patterns, search fails along this path and the node is discarded. We discuss the axiom transformation process later, in the next section we focus on GIST's bottom-up matching process.

6.5.2 Bottom-up Matching

For each possible ISA pattern, we recursively match the IR subtree, node by node, against the ISA pattern (step 4). We treat ISA parameters not fixed by the encoding as variables in ISA patterns. This makes sense as the ISA parameters must be assigned values to generate instances. If a parameter is unassigned (free) in an IR pattern, we consider it to be a parameter to the instruction, and treat it as a symbolic constant (e.g., the **`spTopOffset`** parameter in the previously described **`iadd`** instruction in Section 5.1). This also makes sense as IR instances have all their parameters fixed, thus we can assume that parameters represent actual constants that have been assigned by the compiler front-end. When a free ISA parameter matches an IR tree node, we *bind* it to the value of that IR node, and it influences the remainder of that IR subtree match. This is similar to one-way unification.

GIST enhances the matching process by also considering type information. In addition to comparing tree nodes, we determine if they have compatible CISL types. For example, if we are matching a constant in the IR tree to an immediate parameter on the target, we must determine if the constant can "fit" (in terms of bits and signedness) into the target immediate. If they are compatible, the match succeeds and we continue. If not, we record a *constraint* at this point and continue matching. We return constraints as part of the matching process and they are included along with the solution IS pattern—the compiler adapter can use them in the end to take proper measures (as we mentioned in Section 6.2). After we match an IR subtree, if the destination of the ISA pattern needs allocation (it contains a free parameter), we allocate a scratch location, use it as the destination, and record the fact. In any case, we then replace the matched IR subtree with the destination location of the matching ISA pattern (with or without constraints), and use the resulting tree to form a new search node. States where the (rewritten) IR tree consists of a single **`(seq)`** tree are goal nodes.

In Figure 6.8 we show the bottom-up matching process graphically. The top left tree represents an initial IR pattern (a single assignment) decomposed into all its subtrees that

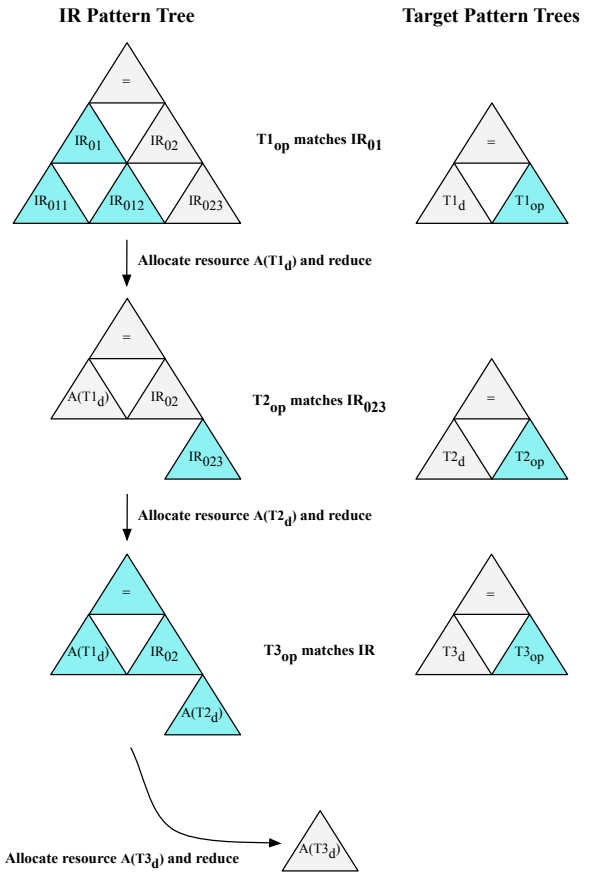


Figure 6.8: IR Subtrees

are to be matched against the target ISA T. We first find a successful match of the target operation $T1_{op}$ to the IR subtree IR_{01}. We allocate $A(T1_d)$, an instance of the target pattern's destination location, and reduce IR_{01} to that location. Next, we find a match between IR_{023} and the ISA operation $T2_{op}$ and reduce the IR subtree to the allocated target destination location $A(T2_d)$. Lastly, a match is found for the remaining IR tree against the target operation $T3_{op}$. We allocate the final destination location $T3_d$ and reduce to match the entire IR pattern. Although there can be several possible match/reduce paths in the search graph that lead to solutions (goal nodes), here we show only one such path for clarity.

To demonstrate the GIST matching process we show a complete trace output generated by GIST during search of a successful match of the store mapped canonicalized form of the JikesRVM Baseline compiler **`iadd`** instruction pattern (introduced in Section 5.1) to a PowerPC pattern sequence. The initial CISL pattern description for **`iadd`** is:

```
S.slot[direct(spTopOffset + 4)] =
  S.slot[direct(spTopOffset)] + S.slot[direct(spTopOffset + 4)];
```

We apply a store map schema mapping the JVM stack to PowerPC main memory, and frame pointer register (referenced in the CISL method **`direct`**) to register 1 on the PowerPC. The resulting canonicalized form is:

```
if (true) M.word[<4+spTopOffset!32>+R[1]] =
  M.word[spTopOffset+R[1]]+M.word[<4+spTopOffset!32>+R[1]];
```

Main memory on the PowerPC (**`M`**) is declared as an addressed store and an index store (**`R`**) specifies the register file. Compiler constants identified during expression reassociation are surrounded by '**`<`**' and '**`>`**' (see Section 5.3.3). We abbreviate parts of the output to conserve space and annotate the important aspects of the IR subtree search process. GIST generates a trace for each path (solutions and failures) in the search graph; the following is an example of the output for one successful path leading to a goal node.

```
Search iadd:
  // We begin with the initial store-mapped canonicalized iadd pattern
  // tree; expressions surrounded by '<' and '>' indicate compiler
```

```
  // constants discovered during expression reassociation:
  if (true) M.word[<4+spTopOffset!32>+R[1]] =
    M.word[spTopOffset+R[1]]+M.word[<4+spTopOffset!32>+R[1]];
  Tree:
  // This is the full interal LISP-like form used internally,
  // annotated with type information (i.e., bu3 indicates a big-endian
  // unsigned CISL bit array 3 bits wide.)
    (seq (gact (true):bit
    (lset
      (lget (mem (sym[M]))
        (add (ccnst (sext (add (iconst (int[4])):bu3
                (param (sym[spTopOffset]))):bs16
              (iconst (int[32])):bu6):bs32):bs32
          (lget (mem (sym[R]))
            (iconst (int[1])):bu5
            (sym[defaultSelector])):bs32)
          (sym[word])):bs32
        (add
          (lget (mem (sym[M]))
            (add (param (sym[spTopOffset])):bs16
              (lget (mem (sym[R]))
                (iconst (int[1])):bu5
                (sym[defaultSelector])):bs32)
            (sym[word]))
          (lget (mem (sym[M]))
            (add (ccnst (sext (add (iconst (int[4])):bu3
                    (param (sym[spTopOffset]))):bs16
                  (iconst (int[32])):bu6):bs32):bs32
              (lget (mem (sym[R]))
                (iconst (int[1])):bu5
                (sym[defaultSelector])):bs32)
            (sym[word]))):bs32):bs32))
  // These are values generated during search including the number
  // of search nodes in the queue (f), the depth (d), the
  // tree size (s), the cost + heuristic value (f(n)),
%% EBM: "the node expansion"? What does that mean -- # of nodes added?
%% TDR: yes, changed to "number of search nodes added"
  // and the current number of search nodes added (e).
  f=0, d=0, s=58, f(n)=0, e=0
  Checking goal tree.
  Checking node expansion limit.
  Checking depth limit.
  Attempting to match.
  // The IR subtree for loading a value from the stack
  // matches the PowerPC lwz instruction; a matching subtree is
  // shown between '{{' and '}}' (we boldface the match for
  // readability).
  Match found:
    if (true) M.word[<4+spTopOffset!32>+R[1]] =
      M.word[spTopOffset+R[1]]+{{M.word[<4+spTopOffset!32>+R[1]]}};
    // The CISL ISA pattern that matched:
    Target instruction lwz: if (true) R[RT] = M.word[D+R[RA]];
    // The ISA tree form that matched:
    Target instruction tree:
```

```
    (seq (gact (true):bit
    (lset (lget (mem (sym[R]))
        (param (sym[RT])):bu5
        (sym[defaultSelector])):bs32
      (lget (mem (sym[M]))
        (add (param (sym[D])):bs16
          (lget (mem (sym[R]))
            (param (sym[RA])):bu5
            (sym[defaultSelector])):bs32):bs32
        (sym[word])):bs32):bs32))
    // The matching process generates bindings from IR
    // constants to target parameters. In this case, the ISA
    // displacement parameter 'D' is bound to the compiler constant
    // <4+spTopOffset!32>, the register index 'RA' is bound to a
    // specific target register '1' (this is the frame pointer that
    // is provided in the store map schema), and the destination
    // register index 'RT' is allocated the symbolic value '@9'.
    // One constraint is generated from this match to indicate that
    // the computed stack offset value must ``fit'' into the 16 bits
    // allowed by the lwz 'D' parameter.
    Bindings   : D => <4+spTopOffset!32>, RT => @9, RA => 1,
%% EBM: How do you indicated signed-ness in FITS?
%% TDR: It would show up as a separate constraint.
%% EBM: Umm, where?
%% TDR: Oh, I see what you mean.  To be honest, I think this is not something I
%% took under consideration with how I modeled the FITS constraint---but, it
%% really should. Or there should be two constraints SFITS and UFITS.
    Constraints: FITS(<4+spTopOffset!32>,16)

Search iadd:
  // Matching continues with the previously matched subtree
  // replaced by the allocated destination register 'R[@9]'
  if (true) M.word[<4+spTopOffset!32>+R[1]] =
    R[@9]+M.word[spTopOffset+R[1]];
  Tree:
  // ...
  f=8, d=1, s=44, f(n)=49, e=12
  Checking goal tree.
  Checking node expansion limit.
  Checking depth limit.
  Attempting to match.
  Match found:
    // Another match to the PowerPC lwz instruction is
    // found on the IR subtree loading a value from a different
    // stack location.
    if (true) M.word[<4+spTopOffset!32>+R[1]] =
      R[@9]+{{M.word[spTopOffset+R[1]]}};
    Target instruction lwz: if (true) R[RT] = M.word[D+R[RA]];
    Target instruction tree:
    // ...
    // The match generates bindings for the diplacement
    // 'D' ISA parameter to the 'spTopOffset' IR constant, 'RA' is
    // bound to '1', and the destination index 'RT' is allocated
    // location '@21'. Again, a fits constraint is generated for
```

```
    // the 'spTopOffset' .
    Bindings   : D => spTopOffset, RT => @21, RA => 1,
    Constraints: FITS(spTopOffset,16)
%% EBM: @21, @9, etc., are obscure; whay not more sequential?
%% TDR: I keep a global allocation counter that is shared by all the search
%% nodes.  I could have kept a separate counter for each, but I was trying to
%% conserve space---probably would not make a big difference.
Search iadd:
  // The resulting partial solution IR subtree is shown
  // with the matched subtree replaced by the allocated destination
  // location 'R[@21]'
  if (true) M.word[<4+spTopOffset!32>+R[1]] =
    R[@21]+R[@9];
  Tree:
  // ...
  // Note this partial solution is only at depth 2 in the
  // search graph with 154 nodes expanded (partial solutions).
  f=29, d=2, s=37, f(n)=47, e=154
  Checking goal tree.
  Checking node expansion limit.
  Checking depth limit.
  Attempting to match.
  // Another match is found for the addition of the two
  // top values of the stack (now in registers 'R[@21] and 'R[@9]')
  // against the PowerPC register-register add instruction.
  Match found:
    if (true) M.word[<4+spTopOffset!32>+R[1]] =
      {{R[@21]+R[@9]}};
    Target instruction add: if (true) R[RT] = R[RB]+R[RA];
    Target instruction tree:
    // ...
    // Bindings are generated for the source registers 'RB'
    // and 'RA' and a new ISA location '@180' is allocated for
    // register index 'RB'. No constraints are generated.
    Bindings   : RB => @9, RT => @180, RA => @21,
    Constraints:

Search iadd:
  // The IR subtree reduced to the allocated destination
  // register 'R[@8]'.
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[@180];
  Tree:
  // ...
  f=29, d=3, s=30, f(n)=45, e=207
  Checking goal tree.
  Checking node expansion limit.
  Checking depth limit.
  Attempting to match.
  // A final match is found on the remaining unmatched
  // IR tree that pushes the result of the addition onto the
  // stack. This matches an entire PowerPC stw instruction pattern.
  Match found: {{if (true) M.word[<4+spTopOffset!32>+R[1]] = R[@180]}};
    Target instruction stw: if (true) M.word[D+R[RA]] = R[RS];
    Target instruction tree:
```

```
    // The correct bindings are generated along with the
    // constraint.
    Bindings   : D => <4+spTopOffset!32>, RS => @180, RA => 1,
    Constraints: <4+spTopOffset!32> => Fits(16)

Search iadd:
  // A goal tree is found at depth 4, 224 nodes expanded
  Tree:
    (seq)
  f=29, d=4, s=1, f(n)=0, e=224
  Checking goal tree.
    Goal tree found.
```

We can summarize the resulting target pattern using an assembly-like format with instruction parameters set according to the match:

```
lwz R[@9]   <- M[R[1]+spTopOffset+4]
lwz R[@21]  <- M[R[1]+spTopOffset]
add R[@180] <- R[@9]+R[@21]
stw R[@180] -> M[R[1]+spTopOffset+4]
```

Although we show only one resulting target pattern, search also generates other patterns that implement the same IR. This pattern comes out first because of the efficiency metric (heuristic cost value), which prefers smaller trees, thus leading to smaller target code sequences. Although we have found that matching canonical forms tends to be fast, we further improve search efficiency by caching previous match results. The search can then use the cached matches directly rather than repeating the process. For example, another solution that was found by re-using the stack offset computation stored in register **`R[@52]`**:

```
addi R[@52]  <- R[1]+spTopOffset+4
lwz  R[@80]  <- M[R[@52]]
lwz  R[@110] <- M[R[1]+spTopOffset]
add  R[@111] <- R[@80]+R[@110]
stw  R[@111] -> M[R[@52]]
```

6.5.3 Axiom Transformation

GIST uses semantics preserving axioms for two different purposes: as part of the canonicalization process to help reduce IR and ISA CISL trees into a normal form to improve matching and during search to transform IR subtrees into semantically equivalent forms

that cannot be inferred automatically—this typically involves deeper semantic properties that violate termination, confluence, and axiom admissibility (see Chapter 5). Prior work relied entirely on axiom application during search with little heuristic guidance [Cattell, 1978]. This provided some success, but the problem was simplified in two dimensions: the ISA is described in terms of the IR and the generated IR uses symbols to refer to addressing modes available on the target. Suitable target addressing modes are pre-computed by the compiler's front-end (i.e., the IR is decomposed to conform to target addressing modes a priori), leaving the matching process to handle only an IR's primary operations.

Despite the potential combinatorial explosion in search nodes generated by axiom applications, these simplifications often averted the problem in practice. In other cases, a simple heuristic was applied to terminate a search path if it got too deep. GIST does not restrict the definitions of IR to conform to any ISA addressing modes. It relies on search to discover ISA instructions that match addressing modes exactly. If it can't find a direct match, the IR is decomposed as part of the normal matching process to determine which ISA instructions can be used to compute the address. The second solution GIST found for the **`iadd`** instruction in the previous section shows how an ISA instruction (**`addi`**) can be used to compute the value of a memory address. (This solution could not be found with prior techniques.)

Because, in favor of generality. GIST drops the restrictions imposed by early approaches, the potential for problems introduced by axiom application during search is increased. We reduce this issue by using canonicalization to limit the number of applicable axioms and apply more sophisticated heuristics to bound the search space. We discuss these heuristics in the next section and focus here on the necessity of search-time axiom application. Consider the following simple rule of addition by zero:

$$\texttt{x} \rightarrow \texttt{x + 0}$$

We exclude this rule from our canonicalization set as it violates the termination condition for a complete term rewriting system (i.e., we can apply the rule infinitely many times).

At the same time, it is a useful transformation in the case when a target ISA includes only an add immediate instruction and we are trying to match a register-register move IR operation—without this basic rule, we could not find a matching ISA sequence. Consider the problem of finding a sequence of target instructions on the PDP-8 for the following IR generated by the PQCC compiler [Leverett et al., 1980] (as originally described by Cattell [1978]):

```
ACC = M.val[offset];
```

This IR loads an accumulator register (**ACC**) with a value from memory.[1] The PDP-8 is an interesting test target because of its minimalistic nature (a sensible requirement for its time). For example, the PDP-8 does not have an instruction to load the accumulator directly. Here is abbreviated output of the solution found by GIST:

```
Search load: if (true) ACC[0] = M.val[offset];
  f=0, d=0, s=16, f(n)=0, e=0
  Attempting to match.
  // We apply an axiom...
  Axiom: $x => $x + 0

// Continue search after applying the ``add by zero'' axiom
Search load: if (true) ACC[0] = 0+M.val[offset];
    f=1, d=1, s=19, f(n)=1, e=1
    Attempting to match.
    // We match the CLA (clear accumulator) instruction
    // on the constant 0
    Match found: if (true) ACC[0] = {{0}}+M.val[offset];
      Target instruction CLA: if (true) ACC[0] = 0;
      Bindings   :
      Constraints:

// The constant 0 is reduced to the ISA destination
// location, which is the accumulator (ACC). Note that
// canonicalization has re-oriented the tree.
Search load: if (true) ACC[0] = M.val[offset]+ACC[0];
    f=5, d=2, s=23, f(n)=1, e=12
    Attempting to match.
    // Another match is found on the entire residual IR tree
    // against the TAD (two's complement add) instruction.
    Match found: {{if (true) ACC[0] = M.val[offset]+ACC[0]}};
      Target instruction TAD: if (true) ACC[0] = M.val[offs]+ACC[0];
```

[1]The CISL pattern for this IR was written as described in Cattell's dissertation. It shows how the IR is tailored to the target instruction set (PDP-8) in that it uses identical memories.

```
      // The bindings and constraints are recorded
      Bindings   : offs => offset
      Constraints: offset => Fits(7)

// A solution is found at depth 3 after expanding
// 17 search nodes
Search load:
    Tree:
      (seq)
    f=8, d=3, s=1, f(n)=0, e=17
    Checking goal tree.
      Goal tree found.
```

First, GIST applies the "add by zero" axiom to transform the memory reference into **`M.val[offset]+0`**. Canonicalization re-orients the pattern so that the constant **`0`** appears first: **`0+M.val[offset]`**. Next, we match the clear accumulator (CLA) instruction on the constant **`0`** and reduce it to the accumulator register (the tree is re-oriented again by canonicalization). Lastly, the two's complement addition (TAD) operation matches the complete residual IR tree, binding the ISA parameter **`offs`** with the IR constant **`offset`**, and generating a constraint indicating that **`offset`** must fit in 7 bits. The resulting sequence found by GIST in assembly-like format is:

```
CLR ACC <- 0
TAD ACC <- ACC+M[offset]
```

Although GIST finds the same PDP-8 sequence as Cattell did, our solution is found in instruction order, uses the same matching process throughout, requires only one axiom application, and two direct matches. Cattell's system dealt with two mismatches, applied the same add by zero axiom, and required special handling (fetch decomposition) to load **`0`** into the accumulator. It is worth mentioning that additional axioms (e.g., commutativity) might have also been applied in Cattell's case if the pattern descriptions were written differently.

In this next example, we show how the combination of GISTs canonicalization and search-time axioms work together to find a solution to a PQCC subtraction instruction:

```
ACC = M.val[offset] - ACC;
```

This IR instruction loads the accumulator with the result of subtracting the accumulator's value from a value in memory. The problem in this case is that the PDP-8 does not include

a subtraction or negation operation making it difficult for a target sequence to be found without axiomatic transformations. This is GISTs output for the solution:

```
// GIST begins with a canonicalized pattern tree. Prior work
// required an axiom for translating subtraction into negated addition
Search subacc: if (true) ACC[0] = M.val[offset]+-ACC[0];
  f=0, d=0, s=24, f(n)=0, e=0
  Attempting to match.
  // We apply an axiom for two's complement arithmetic
  Axiom: -$x => ~$x + 1

// Search continues with the transformed canonicalized tree
Search subacc: if (true) ACC[0] = 1+M.val[offset]+~ACC[0];
    f=3, d=1, s=27, f(n)=1, e=3
    Attempting to match.
    // We match the COMA (complement accumulator) instruction
    // against the IR subtree
    Match found: if (true) ACC[0] = 1+M.val[offset]+{{~ACC[0]}};
      Target instruction COMA: if (true) ACC[0] = ~ACC[0];
      Bindings   :
      Constraints:

// The ~ACC subtree is reduced to the COMA destination location,
// which is the accumulator (ACC)
Search subacc: if (true) ACC[0] = 1+M.val[offset]+ACC[0];
    f=15, d=2, s=26, f(n)=1, e=18
    Attempting to match.
    // Another match is found against the the TAD (two's
    // complement addition) instruction
    Match found: if (true) ACC[0] = 1+{{M.val[offset]+ACC[0]}};
      Target instruction TAD: if (true) ACC[0] = M.val[offs]+ACC[0];
      // Bindings and constraints are generated
      Bindings   : offs => offset,
      Constraints: offset => Fits(7)

// The subtree is reduced to ACC and search continues
// on the residual subtree
Search subacc: if (true) ACC[0] = 1+ACC[0];
    f=47, d=3, s=19, f(n)=1, e=66
    Attempting to match.
    // The complete IR tree is matched to the INCA (increment
    // accumulator) instruction
    Match found: {{if (true) ACC[0] = 1+ACC[0]}};
      Target instruction INCA: if (true) ACC[0] = 1+ACC[0];
      Bindings   :
      Constraints:

// A solution is found at depth 4 after expanding
// 74 search nodes
Search subacc:
    Tree:
      (seq)
```

```
f=49, d=4, s=1, f(n)=0, e=74
Checking goal tree.
  Goal tree found.
```

In this example, GIST *starts* with an IR tree that has been canonicalized by the admissible axiom `$x-$y => $x+(-$y)` transforming subtraction into negated addition. Next, a search time axiom for two's complement arithmetic, `-$x => ~$x+1`, is used to transform the tree into **`ACC = M.val[offset]+~ACC+1`** which is then re-oriented into **`ACC = 1+M.val[offset]+~ACC`**. A match against the complement accumulator (COMA) instruction is found and the IR subtree is reduced to **`ACC`**. This is followed by a match against the two's complement arithmetic (TAD) instruction. Lastly, the INCA (increment accumulator) instruction matches the entire residual tree and a solution is found. The resulting PDP-8 sequence found by GIST is:

```
COMA ACC <- ~ACC
TAD  ACC <- ACC+M[offset]
INCA ACC <- ACC+1
```

With the help of the pattern indexer and canonicalization, the solution found by GIST is straightforward: a single axiom application and three direct matches. Previous work applied several axiom transformations (including commutativity of addition) and generated several mismatches before finding a solution. GIST does generate mismatches along other search paths, but it does not try to fix the problem, it simply drops the path from search and deals only with search points representing valid matches, making its approach uniform and efficient.

Our last example comparing our techniques to previous work illustrates GIST's handling of multiple CISL statements and control flow axioms. In this case, GIST must find a PDP-8 sequence for the following PQCC instruction:

```
if (ACC == 0)
  ACC = 1;
```

In this example, the accumulator is set to 1 if the accumulator is 0. To find a PDP-8 sequence, GIST relies on a search time axiom describing control flow semantics: **`if ($x) { $y } => if (!$x) { PC = PC+1 }; $y`**. This rule says that a conditional can be rewritten into a form that "skips" the statements **`$y`** if the condition `$x` is not true. We admit that this axiom fails unless **`$y`** is 1 word (the distance incremented by **`PC = PC+1`**). To fix the problem we need to extend GIST's tree representation to include the notion of labels and code points. This would allow the distance of the skip to refer to a named label that points to the end of the instruction, thus the axiom could then be written as: **`if ($x) { $y } => if (!$x) { PC = `*`inst-end`*` }; $y`**, where ***`inst-end`*** refers to the point after the guarded assignments specified by **`$y`**. Currently, it is a convention in CISL to use **`PC`** as the "program counter" memory. This does not limit the applicability of CISL, however; architectures that name the program counter differently simply require the same axioms with a different name (e.g., **`IP`**). Future versions of CISL hope to remove this restriction by allowing indexed memories to be annotated as being the program counter. Here is GIST's output:

```
Search setacc: if (0==ACC[0]) ACC[0] = 1;
  f=0, d=0, s=20, f(n)=0, e=0
  Attempting to match.
  // We apply the control flow axiom
  Axiom: if ($x) $y => if (!$x) PC = PC+1; $y

// Search continues with the rewritten IR tree
Search setacc: if (!0==ACC[0]) PC[0] = 1+PC[0];
               if (true) ACC[0] = 1;
    Attempting to match.
    // The first CISL conditional statement matches
    // the PDP-8 skip if not equal (SKPNE) instruction
    Match found: {{if (!0==ACC[0]) PC[0] = 1+PC[0]}};
                 if (true) ACC[0] = 1;
      Target instruction SKPNE: if (!0==ACC[0]) PC[0] = 1+PC[0];
      Bindings   :
      Constraints:

// The tree is reduced by eliminating the statement
Search setacc: if (true) ACC[0] = 1;
    f=10, d=2, s=12, f(n)=1, e=16
    Attempting to match.
    // The next statement is matched against the
```

```
    // SETA (set accumulator) instruction.
    Match found: {{if (true) ACC[0] = 1}};
      Target instruction SETA: if (true) ACC[0] = 1;
      Bindings   :
      Constraints:

// A solution is found at depth 3 after expanding
// 21 search nodes
Search setacc:
    Tree:
      (seq)
    f=13, d=3, s=1, f(n)=0, e=21
    Checking goal tree.
      Goal tree found.
```

GIST finds a solution by applying a control flow axiom and matching two PDP-8 instructions. The SKPNE instruction skips the SETA instruction if the value in the accumulator is not 0. The resulting code sequence is summarized by:

```
SKPNE
SETA  ACC <- 1
```

Other rules are important to ensure we cover the IR instructions completely. For example, the target sequence found previously by GIST for **iadd** is constrained to constants that fit into 16 bits (**4+spTopOffset**). Although this is the most efficient sequence (in terms of number of target instructions) it does not help in cases when the constant is larger. For this we need a rule to *decompose* a constant into parts:

P(X):bs32 → ((P(X) !>> 16) << 16) | (P(X) & 0xFFFF)

This rule indicates that a parameter **P(X)** that is big-endian, signed, and 32-bits wide is equivalent to the expression that performs a bitwise OR on the upper and lower 16 bits. This allows a more general ISA sequence to be found. The axiom is not used by canonicalization because it would exclude the more efficient, special case sequences, thus making it inadmissible. We show the application of this search axiom with the following trace output generated by GIST when searching for a PowerPC sequence that implements the JVM **iconst** instruction:

```
Search iconst_m1: if (true) M.word[spTopOffset+R[1]] = i;
  f=0, d=0, s=19, f(n)=0, e=0
  Attempting to match.
  // We apply the constant decomposition axiom
  Axiom: param($x) => (param($x)!>>16)<<16 | param($x)&0xFFFF

// Search continues with the rewritten IR tree.
// Canonicalization identifies two compiler constants:
// <65535&i> and <i!>>16>.
Search iconst_m1: if (true) M.word[spTopOffset+R[1]] =
                    <65535&i>|<i!>>16><<16;
    f=49, d=1, s=33, f(n)=38, e=89
    Attempting to match.
    // The compiler constant <i!>>16> is matched by
    // the load immediate (li) instruction
    Match found: if (true) M.word[spTopOffset+R[1]] =
                    <65535&i>|<i!>>16><<16;
      Target instruction li: if (true) R[RT] = SI;
      // A register is allocated and the signed
      // immediate parameter (SI) is bound to the constant.
      Bindings   : RT => @68, SI => <i!>>16>
      // No constraints!
      Constraints:

// The constant is reduced to the destination location
Search iconst_m1: if (true) M.word[spTopOffset+R[1]] =
                    <65535&i>|R[@68]<<16;
    f=49, d=2, s=33, f(n)=43, e=344
    Attempting to match.
    // The slwi (shift left by immediate) instruction
    // matches a subtree
    Match found: if (true) M.word[spTopOffset+R[1]] =
                   <65535&i>|R[@68]<<16;
      Target instruction slwi: if (true) R[RA] = R[RS]<<MB;
      // The destination is allocated to register
      // @287 and the immediate is bound to the MB parameter
      Bindings   : RS => @68, MB => 16, RA => @287
      // No constraints!
      Constraints:

// The operation is reduced to the destination R[@287]
Search iconst_m1: if (true) M.word[spTopOffset+R[1]] =
                    <65535&i>|R[@287];
    f=49, d=3, s=30, f(n)=45, e=584
    Attempting to match.
    // The ori (or immediate) instruction matches an IR subtree
    Match found: if (true) M.word[spTopOffset+R[1]] =
                   <65535&i>|R[@287];
      Target instruction ori: if (true) R[RA] = D|R[RS];
      // A destination is allocated and D is bound to the
      // compiler constant <65535&i>.
      Bindings   : RS => @287, RA => @517, D => <65535&i>
      // No constraints!
```

```
      Constraints:

// The operation is reduced to the destination R[@517]
Search iconst_m1: if (true) M.word[spTopOffset+R[1]] = R[@517];
    f=49, d=4, s=23, f(n)=43, e=602
    Attempting to match.
    // The residual IR pattern is matched completely by
    // the stw (store word) instruction
    Match found: {{if (true) M.word[spTopOffset+R[1]] = R[@517]}};
      Target instruction stw: if (true) M.word[D+R[RA]] = R[RS];
      Bindings   : RS => @517, RA => 1, D => spTopOffset,
      // No constraints!
      Constraints:

// A solution is found at depth 5 after expanding
// 623 search nodes
Search iconst_m1:
    Tree:
      (seq)
    f=49, d=5, s=1, f(n)=0, e=623
    Checking goal tree.
      Goal tree found.
```

In this example, GIST uses the constant decomposition axiom to split the **iconst** immediate (32 bits wide) into pieces in order to find a PowerPC sequence that is not constrained by the size of its value. The matches that are found are further improved by GIST's ability to identify compiler constants. It matches half of the immediate against the PowerPC load immediate (**li**) instruction followed by a shift left by immediate (**slwi**) instruction. The remaining part of the immediate is taken care of by matching the or immediate (**ori**) instruction and then the residual IR tree matches the store word (**stw**) instruction completing the search. The solution sequence it finds is:

```
li   R[@68]  <- i!>>16
slwi R[@287] <- R[@68]<<16
ori  R[@517] <- R[@287]|<65535&i>
stw  R[@517] -> M[R[1]+spTopOffset]
```

This ISA pattern is one of several generated by GIST for **iconst**. It was selected as the most general sequence because it was not restricted by any constraints on the immediate parameter **i**. The shortest sequence it found was a load immediate (**li**) followed by a store, however, the immediate **i** was constrained to fit in 16 bits. Because the types of the

matched parameters are part of the generated IS patterns, the adapters can use this information to generate sensible instruction selector code. For example, the load immediate in the most general pattern found above attaches type information to the compiler constant to indicate that it is a 16 bit CISL value. This allows the adapter to take proper measures to ensure it is represented appropriately in compiler implementation code. Multiple patterns, with different constraints, are handled correctly using constructs in the compiler's implementation language (e.g., if-then-else, switch).

To demonstate GIST's unique ability to handle complex IR semantics with grace we show a final example that discovers a target sequence for the JVM **`iaload`** instruction as implemented by the Jikes Baseline compiler:

```
// bounds check
T[1] = S.slot[direct(spTopOffset)];
T[0] = S.slot[direct(spTopOffset+4)];
T[2] = H.cell[arrayRef(T[0], -4)];
if ((type uword_t) T[2] <= (type uword_t) T[1]) {
  throw ArrayIndexOutOfBoundsException
}

// load
T[1] = T[1] << 2;
T[2] = H.cell[regIndexed(0,1)];
S.slot[direct(spTopOffset + 4)] = T[2];
```

This instruction loads an integer value from an index in a Java array. The first value popped off the stack is the array index, this is stored into the Jikes register **`T[1]`**. The second value popped off the stack is the array reference and is stored in **`T[0]`**. Next, we need to retrieve the *length* of the array to perform a bounds check against the index. On Jikes for the PowerPC, the array length is stored in a 4 byte word before the array reference on the heap. This is reflected in the CISL code, storing the length value in **`T[2]`**. An unsigned comparison is performed on the array index (**`T[1]`**) and the length of the array (**`T[2]`**) and an exception is thrown if the length is less than or equal to the index. A left shift is performed on the index to convert the index into an offset and the integer value is loaded

from the heap (**`H.cell[regIndexed(0,1)]`**). Lastly, the integer is pushed on to the stack. GIST finds a solution using as shown in the following output trace:

```
Search iaload:
  if (true) R[4] = M.word[spTopOffset+R[1]];
  if (true) R[3] = M.word[<4+spTopOffset!32>+R[1]];
  if (true) R[5] = M.word[<-4!32>+R[3]];
  if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException;
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
  f=0, d=0, s=168, f(n)=0, e=0

  Attempting to match.
  // GIST matches the first pop to load word (lwz)
  Match found:
  {{if (true) R[4] = M.word[spTopOffset+R[1]]}};
  if (true) R[3] = M.word[<4+spTopOffset!32>+R[1]];
  if (true) R[5] = M.word[<-4!32>+R[3]];
  if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException;
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
    Target instruction lwz: if (true) R[RT] = M.word[D+R[RA]];
    Bindings   : RT => 4, RA => 1, D => spTopOffset,

Search iaload:
  if (true) R[3] = M.word[<4+spTopOffset!32>+R[1]];
  if (true) R[5] = M.word[<-4!32>+R[3]];
  if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException;
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
  f=5, d=1, s=146, f(n)=151, e=8

  Attempting to match.
  // GIST matches the second pop to load word (lwz)
  Match found:
  {{if (true) R[3] = M.word[<4+spTopOffset!32>+R[1]]}};
  if (true) R[5] = M.word[<-4!32>+R[3]];
  if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException;
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
    Target instruction lwz: if (true) R[RT] = M.word[D+R[RA]];
    Bindings   : RT => 3, RA => 1, D => <4+spTopOffset!32>,

Search iaload:
  if (true) R[5] = M.word[<-4!32>+R[3]];
  if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException;
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
```

```
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
  f=9, d=2, s=117, f(n)=127, e=13

  Attempting to match.
  // Loading the array length from the heap is
  // matched to the load word (lwz) instruction
  Match found:
  {{if (true) R[5] = M.word[<-4!32>+R[3]]};
  if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException;
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
    Target instruction lwz: if (true) R[RT] = M.word[D+R[RA]];
    Bindings   : RT => 5, RA => 3, D => <-4!32>,

Search iaload:
  if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException;
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
  f=13, d=3, s=90, f(n)=105, e=18

  Attempting to match.
  // The bounds check matches against the twlle
  // PowerPC instruction. This instruction traps if the
  // comparison is less than or equal.
  Match found:
  {{if (R[5]<=R[4]) throw ArrayIndexOutOfBoundsException}};
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
    Target instruction twlle:
      if (R[RA]<=R[RB]) throw ArrayIndexOutOfBoundsException;
    Bindings   : RB => 4, RA => 5,

Search iaload:
  if (true) R[4] = R[4]<<2;
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
  f=28, d=4, s=74, f(n)=94, e=34

  Attempting to match.
  // The offset computation is matched against
  // the slwi (shift left by immediate) instruction.
  Match found:
  {{if (true) R[4] = R[4]<<2}};
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
    Target instruction slwi: if (true) R[RA] = R[RS]<<MB;
    Bindings   : MB => 2, RA => 4, RS => 4,

Search iaload:
  if (true) R[5] = M.word[R[3]+R[4]];
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
```

```
  f=40, d=5, s=56, f(n)=81, e=47

  Attempting to match.
  // The array access matches against the
  // lwzx load instruction.
  Match found:
  {{if (true) R[5] = M.word[R[3]+R[4]]};
  if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
    Target instruction lwzx:
      if (true) R[RT] = M.word[R[RA]+R[RB]];
    Bindings   : RB => 3, RT => 5, RA => 4,

Search iaload: if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5];
  f=44, d=6, s=30, f(n)=60, e=52

  Attempting to match.
  // Lastly, the push is matched against a store word (stw).
  Match found: {{if (true) M.word[<4+spTopOffset!32>+R[1]] = R[5]};
    Target instruction stw: if (true) M.word[D+R[RA]] = R[RS];
    Bindings   : RA => 1, D => <4+spTopOffset!32>, RS => 5,

// A solution is found at depth 7 after expanding
// 69 search nodes.
Search iaload:
  Tree:
    (seq)
  f=49, d=7, s=1, f(n)=0, e=69
  Checking goal tree.
    Goal tree found.
```

This example shows GIST's ability to handle complex IR semantics that involve several guarded effects, exceptional conditions, and details that model run-time state (array length). The stack access and array length semantics are matched by the PowerPC load word (**lwz**) instruction. The bounds check semantics are handled by the **twlle** instruction. The reason this matches is because our store map schema maps the **ArrayOutOfBoundsException** to a trap on the PowerPC. Additionally, because we are performing an unsigned comparison against the array length (**<=**), both the *length ¿= index* and *index ¡ 0* cases are accounted for (unsigned comparison captures both of these conditions). The array offset computation is matched against the shift left (**slwi**) instruction and the array access matches the **lwzx** instruction. The final push is implemented by the a PowerPC store word (**stw**). The summary of GIST's solution is shown in the following assembly-like listing:

```
  lwz   R[4] <- M[R[1]+spTopOffset]
```

```
lwz   R[3] <- M[R[1]+spTopOffset+4]
lwz   R[5] <- M[R[3]+(-4)]
twlle R[5] <= R[4]
slwi  R[4] <- R[4] << 2
lwzx  R[5] <- M[R[4]+R[3]]
stw   R[5] -> M[R[1]+spTopOffset+4]
```

6.5.4 Pattern Generation

After GIST finds matching ISA pattern sequences for each IR instruction, the results are summarized and generated in a compiler independent format (XML) as instruction selector patterns. Each instruction selector pattern records the IR pattern and matching ISA sequence pattern. The IR pattern lists its parameters along with any constraints that were found during the matching process. The ISA pattern sequence is associated with the parameter bindings that occurred during the matching process. The output patterns are then used by an adapter to generate compiler specific instruction selector code. We describe this process in the next chapter.

6.5.5 Heuristics

As we mentioned previously, GIST relies on a small set of heuristics that are used to guide the search process. These heuristics are based on the size of the pattern tree, the length of the target sequence generated so far, the number of axioms applied, and the number of constraints generated. Here we summarize each heuristic:

Tree Size

The size of the pattern tree is the most important criterion used to determine progress during search. Smaller pattern trees indicate that we are closer to reaching a goal state. Thus, choosing the smallest pattern tree out of all the current partial solutions will lead GIST toward finding a solution quicker.

Sequence Length

There are several possible performance metrics that can be used to determine efficient code sequences. These include the length of the ISA pattern, the code density (i.e., how much

memory space does the generated code occupy), power consumption, and others. In this work, we chose target sequence length to be the most important performance criterion. Thus, the shorter the sequence, the more efficient the IS pattern is. This heuristic can easily be replaced by anything that can gauge efficiency.

Axiom Application

Axioms are typically applied when a match cannot be found for an IR subtree. However, GIST applies axioms to all subtrees at each search point leading to an abundance of derived trees that are often not useful. To control this behavior, we penalize IR trees that have been generated by search-time axioms. This makes partial solutions with several axiom applications less preferable while still allowing trees transformed by a few axioms to be considered. This heuristic works with the tree size heuristic: applying axioms typically makes a tree bigger, thus farther away from a potential goal state.

Generated Constraints

It is important that GIST discover not only IS patterns that cover all possible IR instructions, but also that it find those that are more efficient. Typically, more efficient target sequences are constrained by the their input parameters. To balance the search space in both directions, we penalize partial solutions that have generated constraints during the matching process. This allows more general target sequences to benefit and highly constrained sequences to be favored less. This heuristic works in conjunction with the sequence length heuristic: a target sequence that is shorter is more preferable, however, it is typically constrained in at least one parameter.

These heuristics are computed as a weighted sum and used by GIST to determine the most preferable next search state to visit. We have had considerable success using these four heuristics and in Chapter 8 we present a study on the behavior of search with a variety of weights imposed on each heuristic measure.

6.6 Summary

In this chapter we described GIST's heuristic search procedure. We presented an overview of the search process and each of the important components used by GIST. We described instruction selector patterns and how they relate to semantic descriptions written in CISL. We introduced pattern indexing and discussed its advantage during search and matching. This was followed by an overview of resource allocation and how it contributes in the matching process. Lastly, we described the details of GIST's heuristic search process including search strategy, how candidate ISA patterns are found, the bottom-up matching process, the axiom application process, and IS pattern generation. We finished by presenting the four heuristics used to guide GIST during the search process.

CHAPTER 7

INSTRUCTION SELECTOR ADAPTERS

A novel aspect of GIST is that it can generate instruction selector patterns independent of a specific compiler framework and target machine. Previous work assumes a fixed IR and IS algorithm [Cattell, 1978, Dias, 2008]. Although this allowed research to progress in this area by simplifying the problem, it makes it difficult to apply results in alternative compiler settings. In the previous chapter we discussed how GIST discovers IS patterns from CISL descriptions of the IR and target machine: we use heuristic search to match IR subtrees to ISA patterns and generate as output implementation-independent IS patterns. In this chapter we illustrate how to use a *compiler adapter* to translate GIST-generated output patterns into instruction selector implementation code.

7.1 Overview

The hardest part of implementing an instruction selector—and the most difficult to get right—is choosing which target patterns to use for each IR instruction. GIST generates correct IS mappings assuming that the CISL code describes correct semantics and the axioms are valid. Although not important for this work, CISL is used to generate functional simulators that are capable of executing real programs. This allows the correctness of CISL instruction semantics to be verified by observing the behavior of an executing program. As mentioned previously, we believe axioms can be verified by translating them into Boolean formulae suitable for input into a SAT solver (however, most are simple and easy to verify by inspection). Although we automate the difficult part of IS implementation, we must still translate the generated IS patterns into real IS code. This translation process is accom-

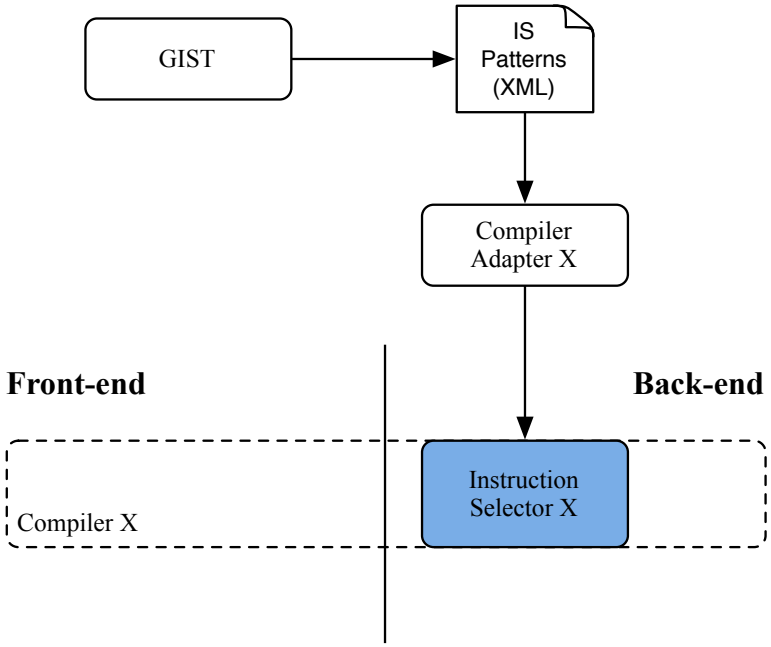


Figure 7.1: Compiler Adapter

plished by a compiler adapter program that generates "code" for a specific IS. For the GIST approach to be successful, the adapter must be significantly easier to implement, extend, and maintain than an IS itself.

In Figure 7.1 we show where the compiler adapter fits in the GIST approach. It is important to observe that a compiler adapter is dependent only on the host compiler framework, *not* on the target machines. Thus, just one adapter is required for each compiler to generate instruction selector code. For example, if we use GIST to generate IS patterns for compiler X targeting the PowerPC and compiler adapter X to translate the patterns into instruction selector X code, we can use the same adapter to generate an instruction selector for GIST-generated patterns targeting a different architecture (e.g., ARM, MIPS) without change. The rest of this chapter describes two compiler adapter implementations for two different compiler frameworks with very different IS strategies: the Jikes RVM Baseline compiler [Alpern et al., 2005] and the LCC C compiler [Hanson and Fraser, 1995].

7.2 Jikes RVM Baseline Compiler Adapter

The Jikes RVM Baseline compiler is a just-in-time (JIT) compiler with a manual instruction selector implementation, written in Java, that translates JVM bytecode directly into target machine code. For each bytecode it has an "emit" method that performs the translation. The Baseline instruction selector provides target-specific support code for generating target instructions directly into memory. For the PowerPC implementation of Baseline the stack pointer is maintained implicitly by the integer variable **`spTopOffset`** and updated at compile time to generate an offset from the frame pointer used at run time. Here is an example: the emit method for the **`iadd`** instruction targeting the PowerPC:

```
// current offset of the sp from fp
public int spTopOffset;

/**
 * Emit code to implement the iadd bytecode
 */
@Override
protected final void emit_iadd() {
  popInt(T0);
  popInt(T1);
  asm.emitADD(T2, T1, T0);
  pushInt(T2);
}
```

This calls support routines that implement popping and pushing an integer from the runtime stack. The **`popInt`** method generates a load instruction and updates the **`spTopOffset`** instruction selector state variable. The **`pushInt`** method works similarly. Temporary registers (e.g., **`T0`**) are used to store intermediate results as we described before, and the **`emitADD`** instruction generates machine code for the PowerPC register-register add instruction using the specified temporary scratch registers.

JVM instructions that require input parameters are implemented in Baseline using method parameters. For example, the **`bipush`** bytecode pushes a byte value onto the stack. The Baseline emit method for **`bipush`** takes an input parameter that is used to generate target machine code:

```
@Override
protected final void emit_bipush(int val) {
  asm.emitLVAL(T0, val);
  pushInt(T0);
}
```

In this example, the 32-bit Java integer **`val`** holds the byte value and is used to generate code to load the value into a scratch location (**`emitLVAL`**) and subsequently to push it onto the stack.

Because some of the JVM bytecodes describe high-level semantics, the corresponding emit routines are not as straightforward. For example, **`getfield`** retrieves the value stored in an object's field and requires a different target implementation depending on the JVM type of the field (e.g., integer, byte, object reference). Thus, the **`getfield`** bytecode is really several different instructions folded into one. These semantics are described in the Jikes CISL description by decomposing the bytecode into parts that represent machine-like instructions that can be handled by CISL and matched by GIST. It is important to mention that this is not a problem for other compilers that deal with lower-level IR representations (e.g., LCC) as the higher-level semantics have already been broken down into machine-like IR instructions by the front-end. It is also worth saying that GIST is the first system that is capable of discovering IS patterns for higher-level IR semantics such as JVM bytecode.

The adapter program needs to translate IS patterns generated by GIST into Java code implementing the emit routines described above. Our Baseline adapter implementation is 230 lines of commented Java code and 255 lines of string templates [Parr, 2009] used to generate parts of the emit methods. The templates are used to map GIST patterns into Baseline implementation code. For example, each target instruction is associated with a template to generate Jikes machine code emit routines as shown in the following excerpt taken from the template file:

```
...
srw(RA,RS,RB)    ::= "asm.emitSRW($RA$, $RS$, $RB$);"
add(RT,RA,RB)    ::= "asm.emitADD($RT$, $RA$, $RB$);"
adde(RT,RA,RB)   ::= "asm.emitADDE($RT$, $RA$, $RB$);"
```

```
subf(RT,RA,RB)   ::= "asm.emitSUBFC($RT$, $RA$, $RB$);"
subfze(RT,RA)    ::= "asm.emitSUBFZE($RT$, $RA$);"
subfe(RT,RA,RB)  ::= "asm.emitSUBFE($RT$, $RA$, $RB$);"
extsb(RA,RS)     ::= "asm.emitEXTSB($RA$, $RS$);"
bclr_special()   ::= "asm.emitBCLR();"
mtlr(RS)         ::= "asm.emitMTLR($RS$);"
...
```

The template name (on the left) is the same name used by the CISL instruction classes for the target, and the parameters refer to CISL variables used to define instruction fields. The right-hand side of the template is the Java code that is generated by the adapter.

The implicit stack pointer (**`spTopOffset`**) must be updated appropriately for each Baseline emit method. We use templates to indicate how each bytecode's emit method is supposed to update the implicit stack pointer:

```
pe_ldc(inst)    ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_pop(inst)    ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_pop2(inst)   ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
```

Additional templates are used to generate conditional code handling constraints on IS patterns and to represent the complete emit method. For this adapter to target another architecture requires only the templates for the target emit methods (shown above) to be changed. For example, to generate Baseline targeting the x86 we copied the PowerPC templates (removing the PowerPC instruction templates) and added templates for each x86 instruction—this took only a few hours before we were generating Baseline Java code for x86. The following examples show output generated by our Baseline adapter for the **`iadd`** and **`iaload`** bytecodes for the PowerPC:

```
// GIST GENERATED
@Override
protected final void emit_iadd() {
  asm.emitLWZ(T0, 4+spTopOffset, 1);
  asm.emitLWZ(T1, spTopOffset, 1);
  asm.emitADD(T2, T1, T0);
  asm.emitSTW(T2, 4+spTopOffset, 1);
  spTopOffset += BYTES_IN_STACKSLOT;
}
```

```
// GIST GENERATED.
protected final void emit_iaload() {
  asm.emitLWZ(4, spTopOffset, 1);
  asm.emitLWZ(3, 4+spTopOffset, 1);
  asm.emitLWZ(5, -4, 3);
  asm.emitTWLLE(5, 4);
  asm.emitSLWI(4, 4, 2);
  asm.emitLWZX(5, 3, 4);
  asm.emitSTW(5, 4+spTopOffset, 1);
  spTopOffset += BYTES_IN_STACKSLOT;
}

// GIST GENERATED.
@Override
protected final void emit_iconst(int i) {
  if (asm.fits(i,16)) {
    asm.emitLI(3, i)
    asm.emitSTW(3, spTopOffset, 1)
    spTopOffset += BYTES_IN_STACKSLOT;
  }
  else {
    asm.emitLI(3,i>>>16);
    asm.emitSLWI(4, 3, 16);
    asm.emitORI(5, 4, 65535&i);
    asm.emitSTW(5, spTopOffset, 1);
    spTopOffset += BYTES_IN_STACKSLOT;
  }
}
```

In the above Java code, the Baseline scratch registers (e.g., **`T0`**) are replaced by their corresponding integer constant values and the calls to Baseline helper routines (e.g., **`popInt`**) have been replaced with direct calls to generated target instructions into memory. Compiler constants have been properly generated to compute stack offsets and **`spTopOffset`** is properly updated as part of the generated instruction selector pattern's emit method. We used our adapter to generate emit methods for the PowerPC and x86 Baseline instruction selector. The generated code compiled correctly in the Jikes environment—we highlight our results in Chapter 8.

7.3 LCC Compiler Adapter

The LCC C compiler is written in C and uses BURS technology [Fraser et al., 1992] for its instruction selector. As we mentioned in Chapter 2, a set of IS patterns are provided by the compiler implementor and used as input to a bottom-up rewriting process to determine optimal target code sequences. In this case study, we use GIST to generate the IS patterns automatically and an adapter to translate the patterns into the BURS format used by the LCC instruction selector. Here is an excerpt of the LCC BURS description:

```
...
reg: ADDU4(reg,rc)    "addu $%c,$%0,%1\n"  1
reg: BANDI4(reg,rc)   "and $%c,$%0,%1\n"   1
reg: BORI4(reg,rc)    "or $%c,$%0,%1\n"    1
reg: BXORI4(reg,rc)   "xor $%c,$%0,%1\n"   1
reg: BANDU4(reg,rc)   "and $%c,$%0,%1\n"   1
...
```

Each line corresponds to a single IS pattern with the format: *destloc*:*irtree targetasm cost*. The *destloc* is the target destination location the IR tree is reduced to after a successful match, the *irtree* is the IR tree pattern to match, the *targetasm* is the target assembly to generate upon a successful match, and the *cost* is a cost metric for the target instruction. The IR parameters (e.g., **`reg`**, **`rc`**) refer to other BURS patterns representing IR subtrees and are accessed in the target assembly template as **`%0`** for the first parameter, **`%1`** for the second, and so on. Register allocation is handled by the BURS process and can be accessed using the template variable **`%c`** to refer to the allocated resource. Other template variables are used to access constants and evaluate compiler constant expressions at compile-time. The cost metric is used by BURS to determine the quality of a target sequence. These costs are different from the heuristic costs used by GIST during the search process, but relate to its efficiency metric in terms of the length of the target sequence for the IS pattern. We do not pollute CISL descriptions with this information because efficiency depends on the target execution environment—we expect this information to be provided separately or as part of user provided efficiency heuristics in future versions of GIST.

The adapter used to generate LCC BURS rules is similar to the Jikes adapter: it consists of 469 lines of commented Java code (most of this is string related manipulation to convert GIST patterns into BURS-formatted rules) and 618 lines of string templates (a significant portion of this refers to boilerplate BURS-related C code support that is required by LCC's BURS generator program) used to generate the BURS rules for the 32-bit MIPS architecture; the x86 and SPARC template files are similar. The following is an excerpt from our MIPS template file:

```
...
REM(rd,rs,rt)     ::= <<"rem $%c,$%0,$%1\n"   1>>
REMU(rd,rs,rt)    ::= <<"remu $%c,$%0,$%1\n"  1>>
MUL(rd,rs,rt)     ::= <<"mul $%c,$%0,$%1\n"   1>>
MULTU(rs,rt)      ::= <<"multu $%0,$%1\n"      1>>
ADD_R(rd,rs,rt)   ::= <<"addu $%c,$%0,$%1\n"  1>>
ADDI(rd,rs,imm)   ::= <<"addiu $%c,$%0,$%1\n" 1>>
ADDU_R(rd,rs,rt)  ::= <<"addu $%c,$%0,$%1\n"  1>>
ADDIU(rd,rs,imm)  ::= <<"addiu $%c,$%0,$%1\n" 1>>
...
```

Similar to the Jikes templates, the template name (on the left) refers to CISL instruction classes defined for the target and the right-hand side is part of the BURS rule to be generated—other templates are used to generate the appropriate BURS pattern tree for each IR. The adapter's job, in this case, is easier because it does not need to generate CISL parameters as part of the target template as was the case for Jikes; this is handled by BURS independently as part of its register allocation process or inserted directly as part of its internal matching process.

Because LCC's IR is low-level, in most cases GIST found a one-to-one match with a target instruction that was identical to the hand-written BURS file. In some cases, the GIST generated rules lacked sufficient compiler internal state representation (i.e., the instruction selector state was not included as part of the CISL description). For example, the original BURS rule for the **`CVII4`** IR instruction (convert to integer) accesses part of the IR data structure to generate a single rule to handle conversions from shorts and halfs to integers:

```
reg: CVII4(reg)  "sll $%c,$%0,8*(4-%a); sra $%c,$%c,8*(4-%a)\n"  2
```

The hand-written pattern uses the **%a** template variable to access the size of the conversion to compute at compile time the correct shifting distance. GIST generated the conversions directly:

```
reg: CVII4(reg) "sll $%c,$%0,8; sra $%c,$%c,8\n"   2
reg: CVII4(reg) "sll $%c,$%0,16; sra $%c,$%c,16\n" 2
```

The CISL description for LCC can be modified to include compiler state representing the IR's conversion size value and the CVII4 instruction could take advantage of that fact to describe the generic form. Canonicalization would then identify the computation as a compile-time constant and the identical target sequence would be generated by the adapter. We used our LCC adapter to generate BURS rules for the MIPS, x86, and SPARC architectures and successfully built LCC for each case—we highlight our results in Chapter 8.

7.4 Summary

In this chapter we demonstated that generating instruction selectors automatically in a compiler- and target-independent manner is possible using GIST and compiler adapters. We discussed two case studies involving two very different instruction selector implementations for the Jikes RVM Baseline compiler and LCC. We showed that our adapters are small and easy to implement and generate correct output in different compiler frameworks targeting different architectures. Although anecdotal, it is interesting to mention that these adapters were implemented over the course of a summer program by undergraduate students *without* previous compiler experience.

CHAPTER 8

EXPERIMENTS AND RESULTS

8.1 Experimental Infrastructure

We implemented the CISL compiler and GIST in Java and ran all experiments on standard desktop/laptop machines containing Intel's Core 2 Duo and G5 PowerPC. We wrote substantial CISL descriptions for two IRs and four different architectures and a partial description of the PDP-8 and PQCC compiler IR to reproduce Cattell's earlier results.

8.2 Experimental Methodology

We designed our experiments to answer the following questions: For what fraction of IR patterns does GIST find at least one IS pattern for each IR and target combination (coverage)? How long does it take GIST to find instruction selector patterns (IS generator performance)? How fast does a compiler with a GIST generated instruction selector run (IS performance)? And, how fast is the resulting compiled code (execution performance / IS quality)? We use the term *coverage* to measure "usefulness", i.e. how many patterns could GIST find to cover the functionality of the IR. Since selectors translate from IR to target, *IR coverage* is a priority. We are not seeking one-to-one translation in the general case—indeed, it can be detrimental to pursue total coverage for some *targets*. For example, an instruction selector for a JVM to IA32 compiler, does not need to cover IA32 BCD manipulationi instructions.

For GIST to be practical, it must run in time comparable to a typical compiler build. That is, running GIST should not slow down the development of the compiler we are tar-

geting. In the typical case, GIST is executed once to generate the IS patterns, so GIST performance is not as important. However, if we are experimenting with the IR or ISA (i.e., we are adding/removing instructions), GIST performance is more important. Since GIST is designed to be used for real compilers and compiler/architecture exploration, GIST performance is important. We investigate the performance of GIST in eight different IR/ISA IS combinations and compare the execution time in terms of the complexity of the ISA (e.g., number of instructions). We show results for total execution time and how long it took on average for GIST to find an IS pattern.

In addition to GIST performance, we are concerned with the performance of the compiler using a GIST generated IS and the efficiency of the code it generates. Thus, the generated IS should not have a negative impact on the overall execution time of the compiler—it should run in time no worse than a hand-coded (and optimized) IS. In addition, the target code generated by a GIST generated IS must be high-quality: it must have performance that is no worse than than code generated by a hand-coded IS. We compare the execution time of three compiler-framework/ISA pairs: Jikes RVM Baseline compiler targeting PowerPC, `lcc` targeting x86, and `lcc` targeting MIPS. For two of these combinations, Baseline/PowerPC and `lcc`/x86, we show that a GIST generated IS produces code that no worse, and in some cases better, than the original hand-crafted IS. This sample covers both the virtual machine and conventional (front-end/back-end) compiler space along with different targets, and allows us to measure performance.

Additionally, we wanted to understand the behavior of our search procedure with respect to our set of chosen heuristics. For this we designed experiments to answer the questions: Is there an optimal setting of weights that can be applied to each heuristic that can be used across all IR and ISA combinations to produce the best GIST results? For this we compared GIST execution time, coverage, search depth, and node expansion for different weights on pairs of heuristics.

Since GIST is written in Java, we were able to run it on a wide variety of OS and hardware platforms, from laptops to large servers. Running time is variable, depending mostly on the target instruction set (CISC takes longer than RISC), but is in the range of seconds to tens of seconds. Since GIST is used only at compiler build time, we find its performance quite acceptable.

8.3 CISL Results

To show that CISL is capable of describing a variety of IR and architectures we described three substantially different types of IR, `lcc`, Jikes RVM bytecode (both implicit and explicit stack), and a partial description of the PQCC IR, and several architectures including PowerPC, ARM, MIPS, x86, and partial SPARC and PDP-8 descriptions. The ability of CISL to span both compiler IR and architecture ISA is novel in this area of study and is a substantial contribution. We have shown a number of examples of CISL's descriptive capabilities and include additional excerpts from each of the descriptions in Appendix A.

8.4 IS Generation Performance

We show GIST's performance results in Figure 8.1 and Figure 8.2. We show in Figure 8.1 the average execution time of GIST on each IR/ISA combination. The y-axis is the total number of seconds (log scale) that it took GIST to run on an IR/ISA pair. The x-axis shows each of the 8 combinations that were tested. The results show that for CISC-like architectures (e.g., x86), GIST takes the most time (i.e., 7 minutes for `lcc`/IA32 versus 16 seconds for `lcc`/MIPS32). Combinations that include the ARM architecture took more time (2 minutes on average). This makes sense because the ARM provides some CISC-like features. In particular, its shifter operand addressing mode allows data processing instructions to have 16 possible forms that range from immediates to operands that can be shifted by registers.

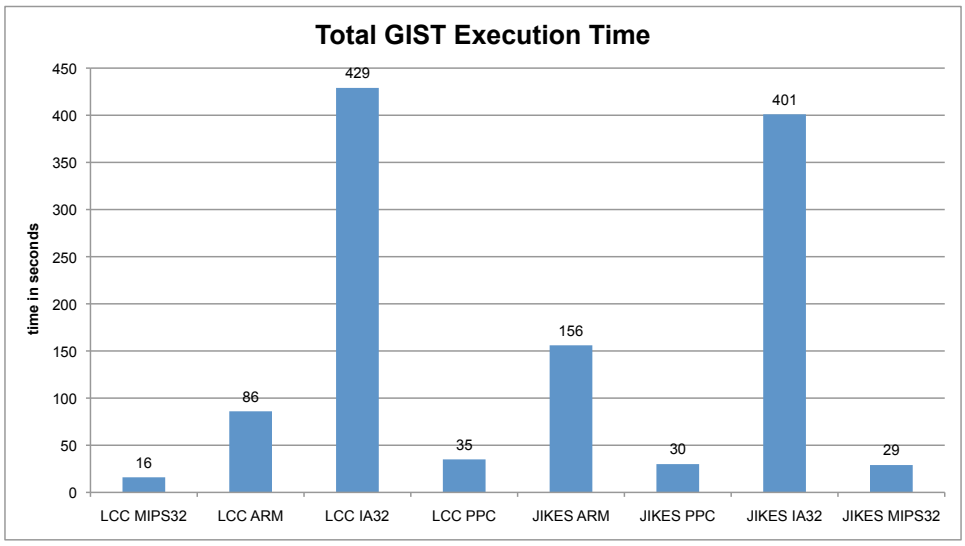


Figure 8.1: Total GIST Performance

We show in Figure 8.2 the average time it takes GIST to find an IR match on the target ISA. Again, these numbers show that the more complicated the ISA (e.g., CISC), the longer it takes for GIST to find a match. The same arguments for ARM also apply. In the best case, it took GIST less than half a second to find an IR match for `lcc`/MIPS32. In the worst case, it took GIST an average of 5 seconds to find an IR match for JIKES/IA32.

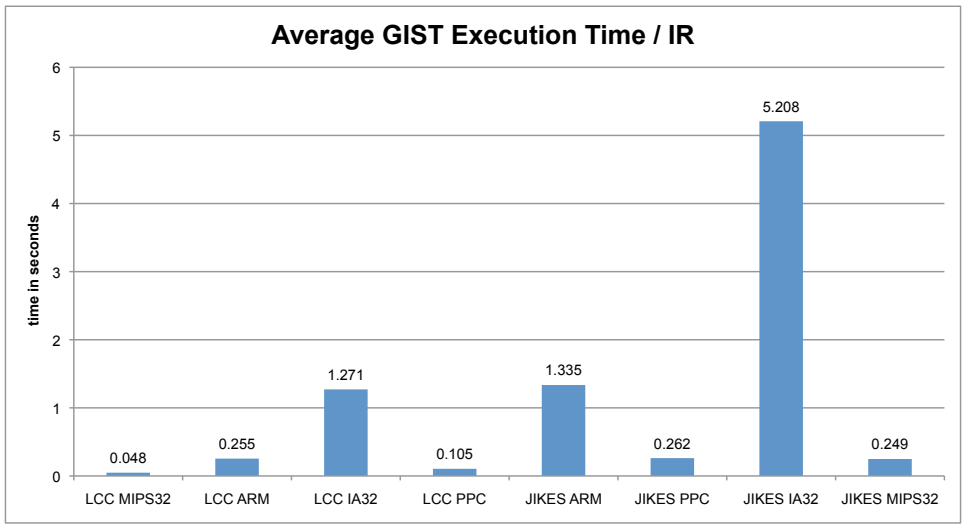


Figure 8.2: GIST Performance per IR

These results show that GISTs performance is good and its use in real compiler development is tolerable. GIST performance could be further improved if it performed subsequent searches based only on the modified parts of CISL IR descriptions. For example, adding an IR operation should only require search for an IS pattern for that IR operation, not running through the entire IR instruction set. This capability currently exists: GIST can be executed on a subset of the IR instructions. To further increase performance, CISL could be pre-compiled to an intermediate format to eliminate parsing and semantic checking time. Regardless, GIST's performance is acceptable in terms of compiler development time.

In Figure 8.3 and Figure 8.4 we show the average[1] search depth (number of nodes along a path) and average number of nodes expanded before GIST reached a solution. In most cases, a solution was found

at an average depth of 2, whereas in other cases (e.g., Jikes/ARM) at depth 4. Although not included on this graph, solutions were found by GIST at a maximum depth of 10 for all the Jikes combinations except the x86 target. This makes sense as these results show only the depth of successful patterns; the longer, more complex, IR trees matching a large and complicated target are very difficult. The second graph shows the average number of nodes expanded for a solution. An expanded node refers to a point in the search graph that has been generated by a match, contains a residual tree, but has not been visited yet. Not surprisingly, for a typical IR (e.g., `lcc`) targeting a RISC-style architecture, the number of nodes remains fairly small, whereas the x86, ARM, and Jikes RVM cases require additional exploration.

8.5 Coverage Results

We provide results for GIST source coverage in Table 8.1, in which columns stand for the source/target combination, number of source operations in the VM/IR specification (*Source*), how many source instructions we effectively covered (*Covered*), the number of

[1] For this average we use the mode (the most common value) to give the best sense of the typical behavior.

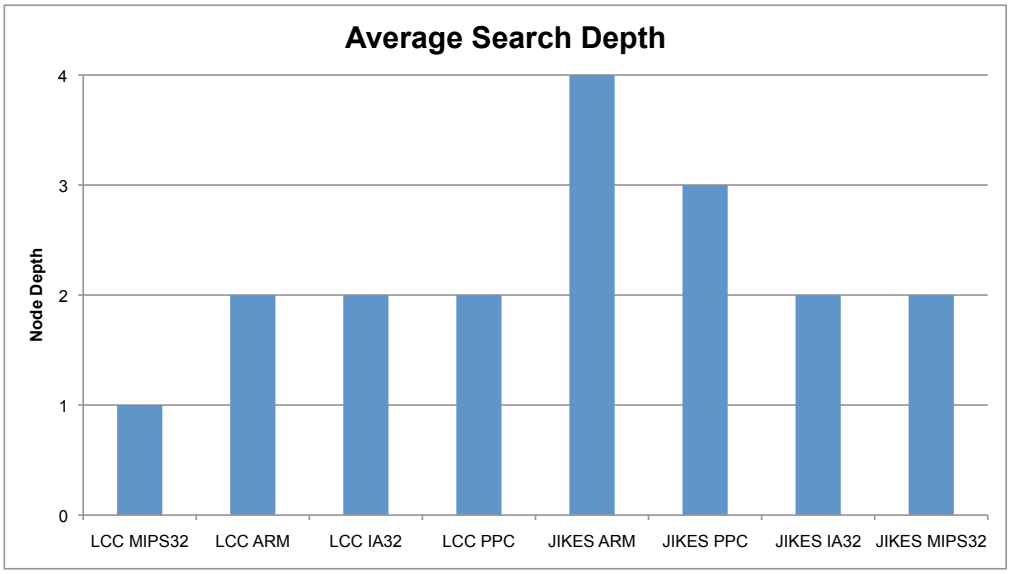


Figure 8.3: Average Search Depth

target instructions (*Target*), and the coverage percentage with respect to the source operations.

Effective operations are the subset of the source IR that we consider for matching. We omit non-effective operators based on various criteria, typically because the source semantics are highly variable, e.g., **`invokevirtual`** has nearly infinite variations in the called method signature; it needs to be broken down into instruction-like pieces, or the operations are not supported on the target architecture, e.g., the Baseline description has 42 bytecodes that depend heavily on the semantics of the run-time system, which we omit from the effective set (we can map them to specific run-time calls). We expect to support run-time calls in the future by describing their interface without having to give all the details of how they are implemented. This will allow an IR to match targets supporting a specific run-time performing the same syntactic matching process. The most common versions of the ARM do not include floating-point or division, which removes $35 + 2 = 37$ operators from matching consideration. Many of these would be implemented instead by library calls. We also currently do not include floating-point operations for the IA-32 instruction set due to

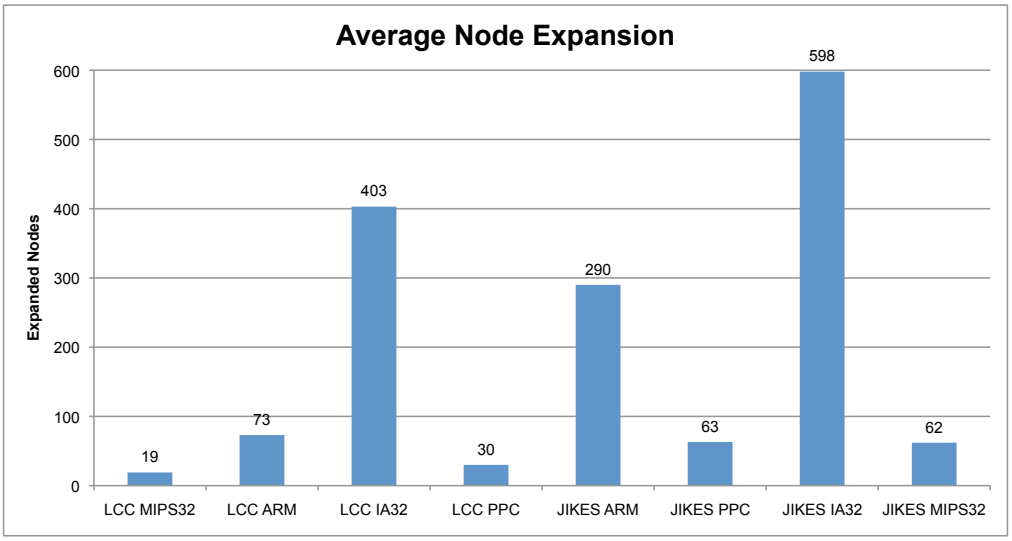


Figure 8.4: Average Number of Search Nodes per IR Instruction

the 64-bit/80-bit representational difference between `lcc` and IA-32 (although we plan to add support for this in the future). This removes 36 operations from the effective set.

The results show some interesting patterns. We see higher matching success rates for the `lcc` BURS specification versus the JVM specification. JVM semantic descriptions are more difficult to match than conventional compiler architectures for two reasons. First, semantic trees produced from JVM specifications are often larger on a per-instruction (or bytecode) basis. We call this the *higher-level of semantics problem*, since each description unit has more semantics packed into it. GIST can handle many of these situations because ISA semantics tend to represent smaller units of functionality, but one must be careful with the heuristics to limit the size of the search space.

Second is the *run-time* problem, where the semantics of the JVM directive depend on the JVM run-time system. We excluded many of these from the effective operation set, and the remainder are the source of most of the unmatched JVM bytecodes. However, we can match them eventually by extending the target description to include available run-time calls. For the moment this remains future work.

	Source	Target	Covered	Percent
LCC MIPS32	227	104	223	99%
LCC ARM	227	589	210	93%
LCC IA32	227	143577	209	93%
LCC PPC	227	96	223	99%
JIKES ARM	96	589	87	91%
JIKES PPC	117	96	114	98%
JIKES IA32	77	143577	33	43%
JIKES MIPS32	117	104	96	83%

Table 8.1: GIST Source Coverage Results

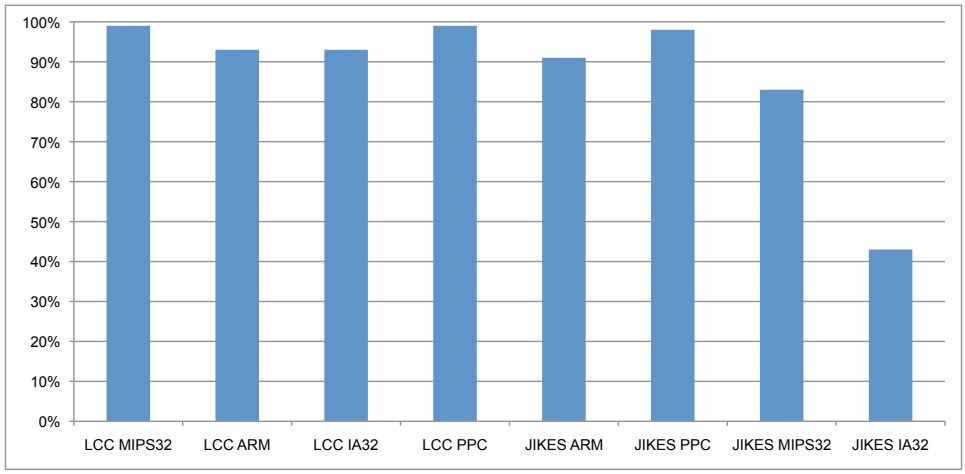


Figure 8.5: GIST Source Coverage Results

From the figures we see lower coverage rates for Jikes MIPS32 and Jikes IA-32. Jikes MIPS32 is still a work in progress: it needs additional store mapping rules and search axioms to attain success rates comparable to `lcc` MIPS32. We include this case because it is an example of how quickly a new combination can start to produce useful results: we covered 83% of the Jikes RVM bytecodes with less than an hour of work using a simple store map and two existing CISL descriptions.

The 43% coverage rate for the Jikes IA-32 combination stems from the the fact that CISL is also used to generate functional simulators. Because of this, our IA-32 description specifies the full set of IA-32 instructions (approximately 400K unique instructions, although we manually restricted the target set to 150K for this test). Pattern discovery is thus orders of magnitude more difficult, although we are able to achieve 93% coverage for `lcc`. However, JVM semantics typically involve larger CISL trees, thus matching them against a target with a large (and complicated) instruction pool explodes the search space. Figure 8.4 shows that the average number of nodes expanded for a *successful* Jikes IA-32 match is 600. It takes on average 5 seconds to find a solution for this combination. We note that we could easily improve performance for this case by reducing the IA-32 instructions to a number (i.e., a few hundred) that is more comparable to the subset used in a typical IA-32 code generator, and in prior instruction selector generator research. In fact, we saw much higher coverage when our IA-32 description was only partially complete. Developing a CISL extension to facilitate automatic subsetting of a large ISA description for different tools is a future research problem. Presently, however, we are indulging our curiosity to see what GIST can discover in instruction sets that are many times larger than those tackled by previous systems—experimentation that is only possible because GIST is so efficient.

8.6 IS Peformance

We compared the execution time of the GIST-generated ISs for `lcc`/MIPS, `lcc`/x86, and Jikes RVM/PPC against their original implementations. In all cases, the run-time perfor-

mance of the GIST-generated instruction selectors was statistically the same as the originals. This is not surprising because our adapter programs for each are producing code that works in pre-existing IS frameworks with identical IS algorithms and data structures.That is, the goal of GIST is not to change the fundamental approach of the instruction selector algorithms, rather it "adapts" the GIST patterns so they can be *used* by those algorithms. For example, the BURS rules generated by GIST for the `lcc` compiler are almost identical to the original hand-written patterns. The Jikes RVM Baseline adapter generates Java code that emits target instructions using a built-in assembler. Unlike the original, the GIST version generates calls the assembler's emit procedures directly rather than using programmer friendly helper routines—which are probably inlined by the Java compiler.

8.7 Generated Code Performance

As stated above, we tested the generated instruction selectors in two compiler frameworks: Baseline targeting PowerPC, and `lcc` targeting MIPS. We compared the running time of code generated from our selector with that generated by the original framework. We used a subset of the DaCapo benchmarks [Blackburn et al., 2006] for Baseline and the `lcc` test suite [Hanson and Fraser, 1995] for `lcc`. These results appear in Figures 8.6, 8.7, and 8.8 as a slowdown ratio (less than one indicates a speedup). While it is comforting to see that we obtain a speedup in many cases, we do not claim that this is statistically significant—note the scale of the graphs. Essentially the performance of the generated selectors does not differ from that of the originals. Note that this process also validates the functionality of the instruction selectors.

8.8 Heuristic Study

To understand better the effect of our chosen heuristics we ran GIST on all IR/ISA combinations with different pairs of heuristics and weights. GIST computes the heuristic value as a weighted sum of the individual heuristic measures: $h(n) = s \times w_1 + l \times w_2 + a \times w_3 + c \times w_4$,

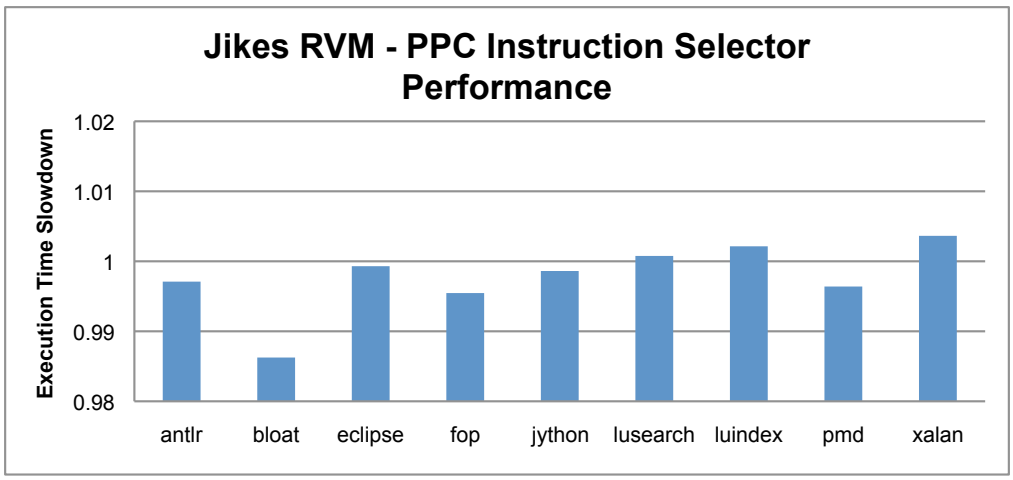


Figure 8.6: Jikes RVM-PPC slowdown (lower = better)

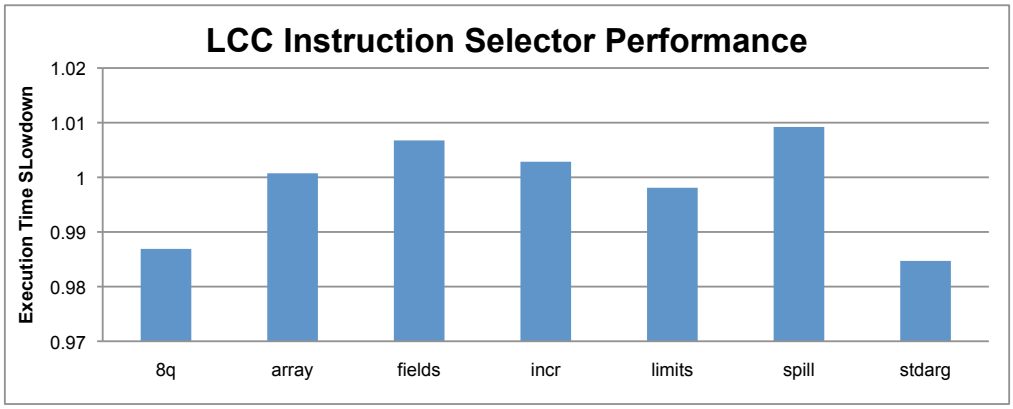


Figure 8.7: `lcc` - MIPS slowdown (lower = better)

where s is tree size, l is target length, a is axiom application, and c is the number of constraints (see Section 6.5.5). We paired *tree size* with *target length* and *axioms* with *constraints*. For each pair (x, y), we fixed the weight of 1.0 to x and then ran 10 GIST runs applying weights from 0.0 to 1.0, in 0.1 increments, to y. We then performed the same set of experiments with y fixed to 1.0 and varied x. The purpose of these experiments was to determine if a setting of the weights exists that can maximize coverage across all IR/ISA combinations.

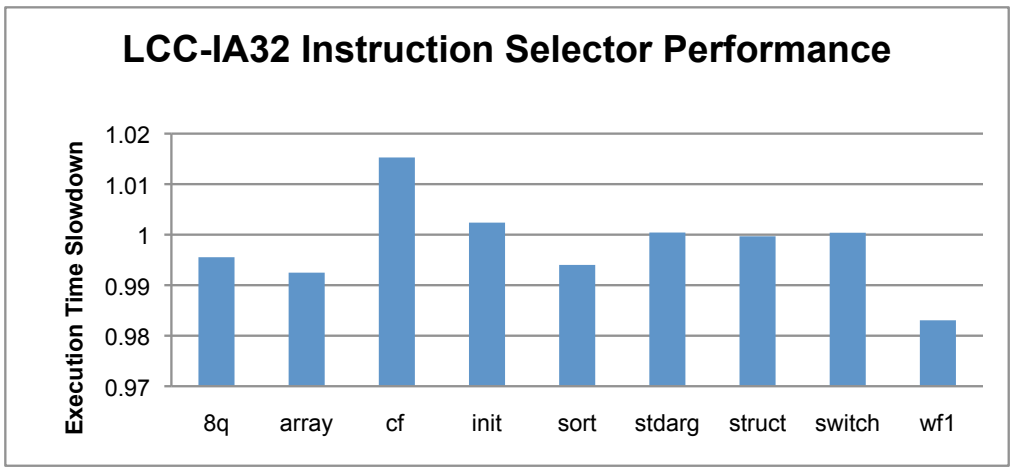


Figure 8.8: `lcc` - IA32 slowdown (lower = better)

The most interesting result is found in Figure 8.9 that clearly shows that a weight of 0.1-0.3 on tree size has a significant impact across all IR/ISA pairs: the coverage is increased by 20% in each case. In most cases, the application of larger weights to tree size worsens coverage—for the Jikes/MIPS case it drops to 40%. In Figure 8.10 we fixed the tree size and varied the target length weight. Although not as drastic as the previous figure, this shows that the higher the weight, the better the coverage. This is especially true for Jikes/PPC as it jumps from 60% to over 80% coverage between a weight of 0.6 to 0.7. Figure 8.11 and Figure 8.12 show results for the axiom/constraint heuristic pair. Not surprisingly, the coverage for each IR/ISA pair are not impacted nearly as much. This makes sense as axioms apply to only a small number of cases (unlike prior work) and work in conjunction with tree size (i.e., axioms tend to increase tree size). Constraints are similar in that they do not limit a solution's being found, rather they limit the type of solution that is found (i.e., special case versus general). In Figure 8.13 we vary the size and fix the axiom heuristic. We notice that coverage increases for each case except for JIKES PPC and JIKES MIPS32. This behavior is similar in Figure 8.15 when we vary the size and fix the constraint heuristic. The decrease in coverage also shows up for the JIKES MIPS32 case in Figure 8.17. This indicates that the complexity of the IR and ISA trees may require different settings for the

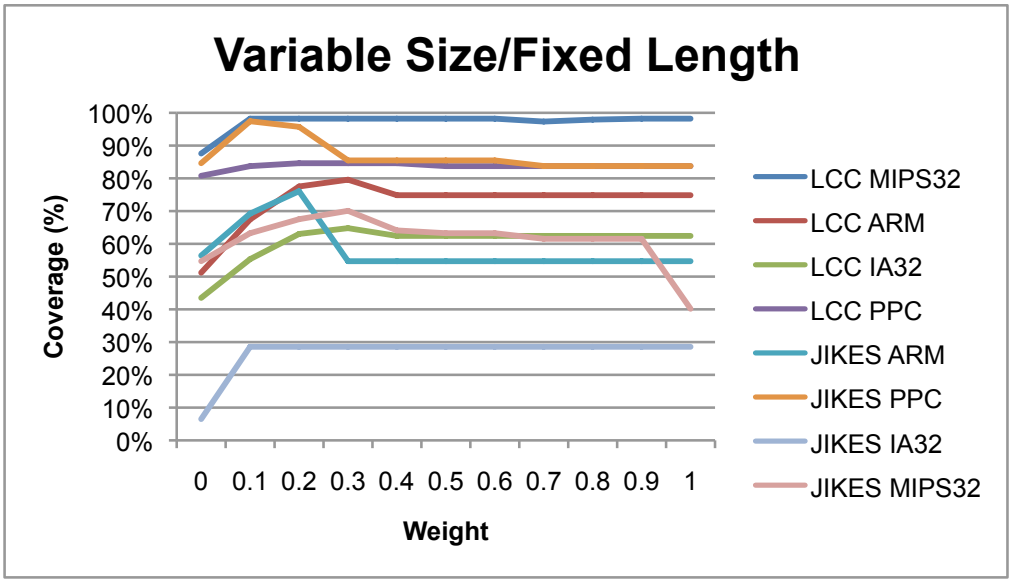


Figure 8.9: Variable Size/Fixed Length

heuristic weights. We could use the size of the original IR tree and the number and size of the ISA to choose settings of the weights that will provide the best possible coverage. The other pairs do not have an impact on IR coverage.

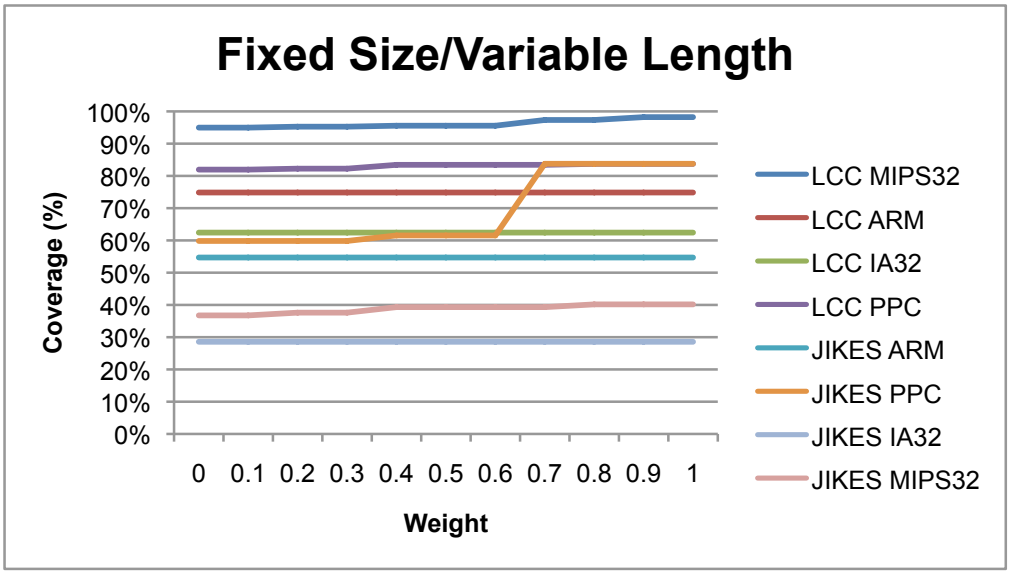


Figure 8.10: Fixed Size/Variable Length

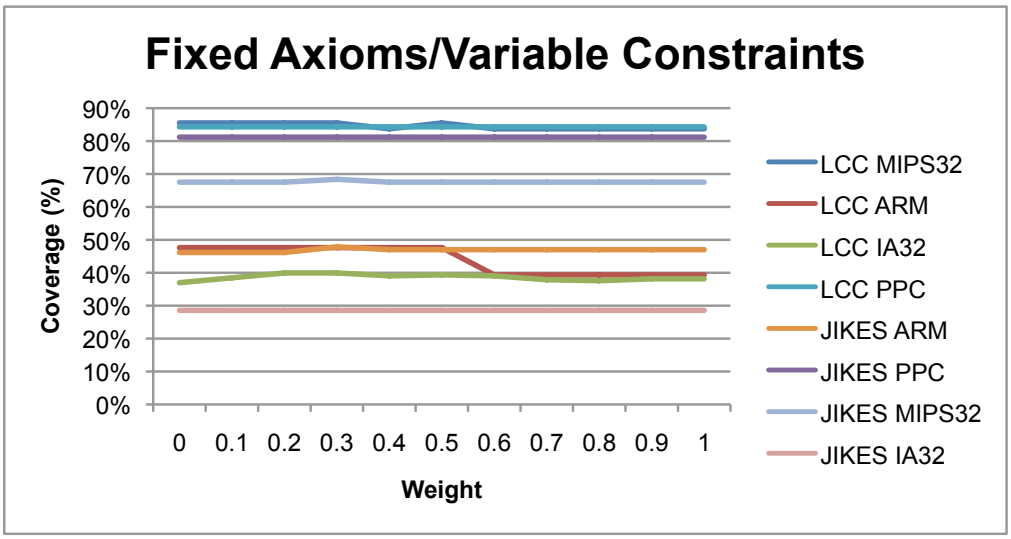


Figure 8.11: Fixed Axioms/Variable Constraints

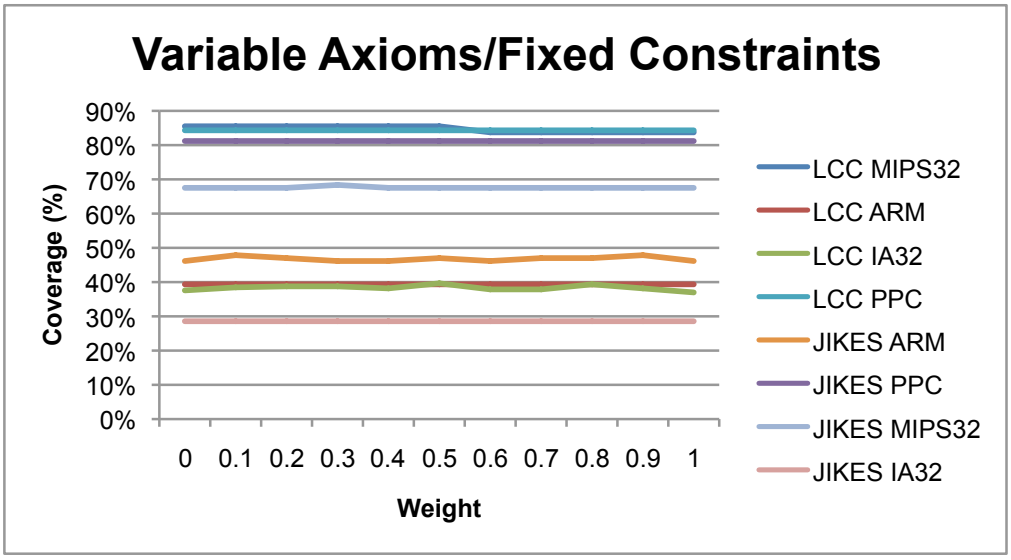


Figure 8.12: Variable Axioms/Fixed Constraints

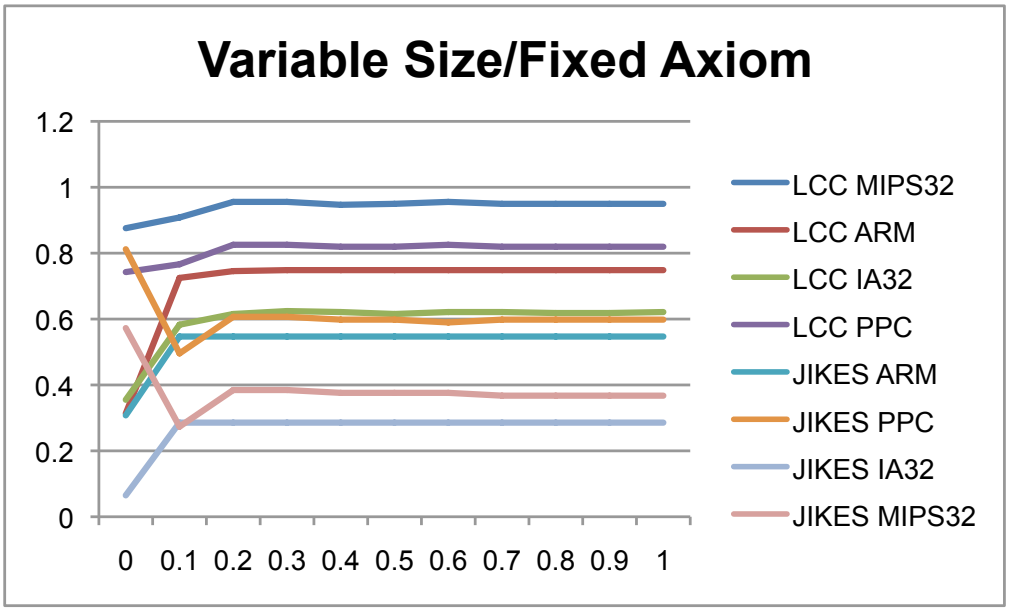


Figure 8.13: Variable Size/Fixed Axioms

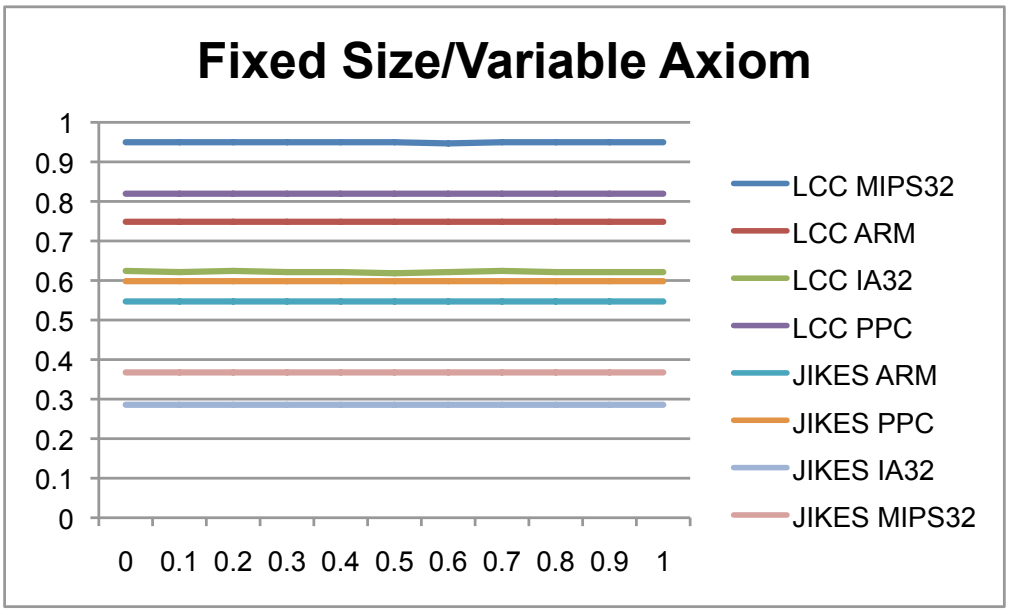


Figure 8.14: Fixed Size/Variable Axioms

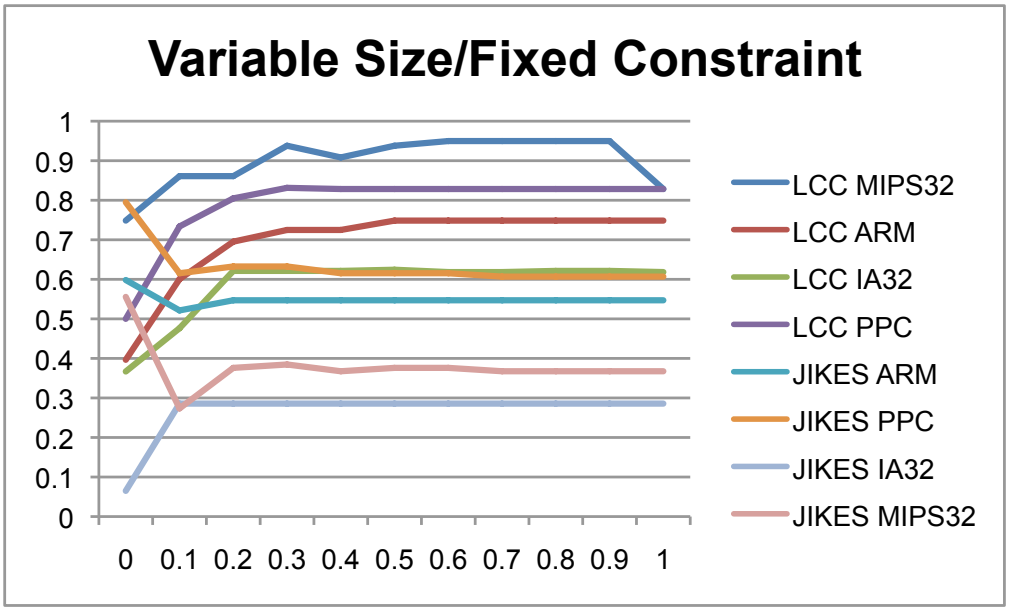


Figure 8.15: Variable Size/Fixed Constraints

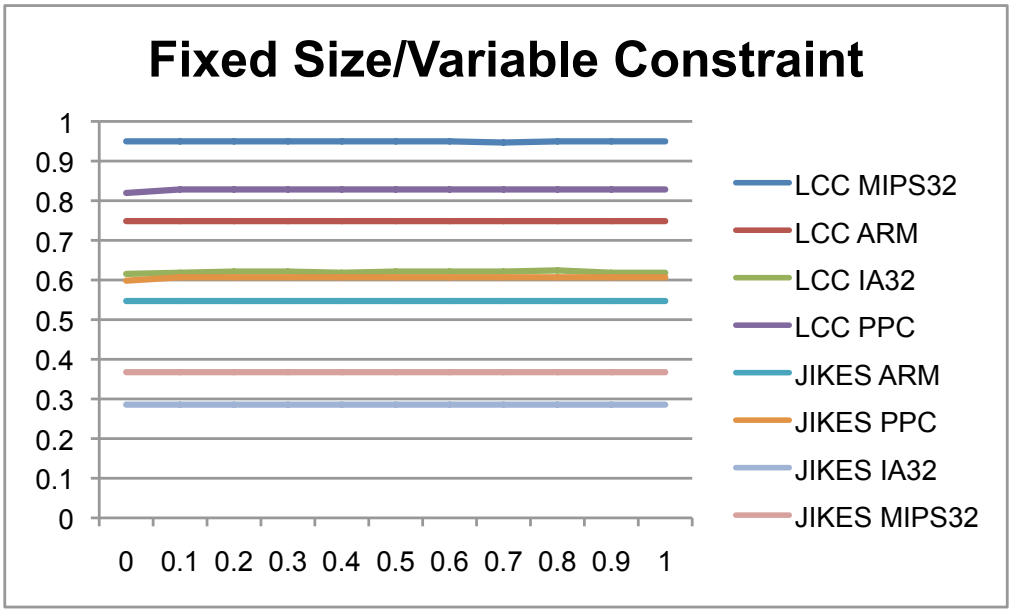


Figure 8.16: Fixed Size/Variable Constrainst

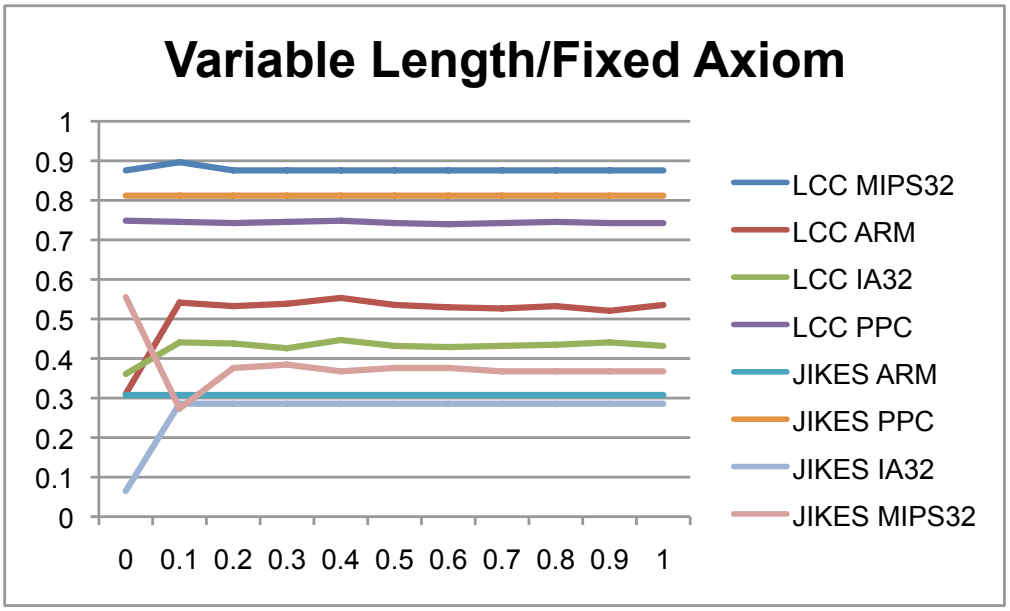


Figure 8.17: Variable Length/Fixed Axioms

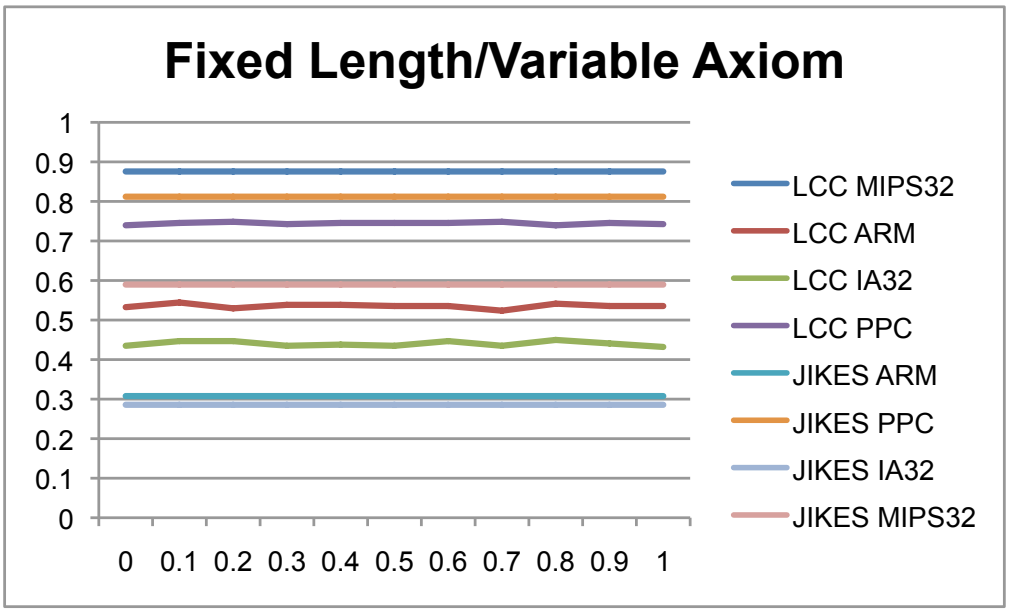


Figure 8.18: Fixed Length/Variable Axioms

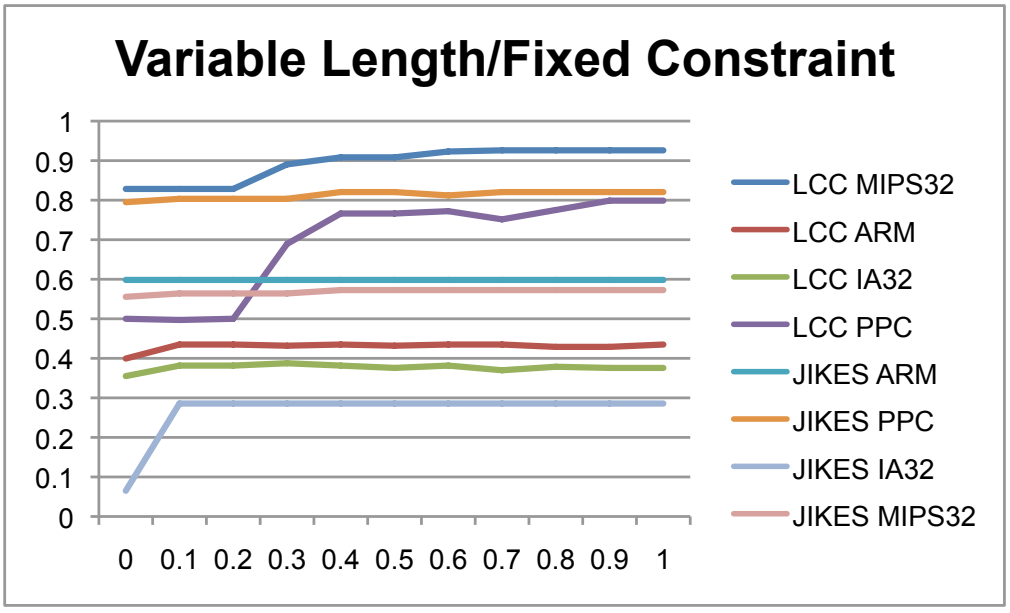


Figure 8.19: Variable Length/Fixed Constraints

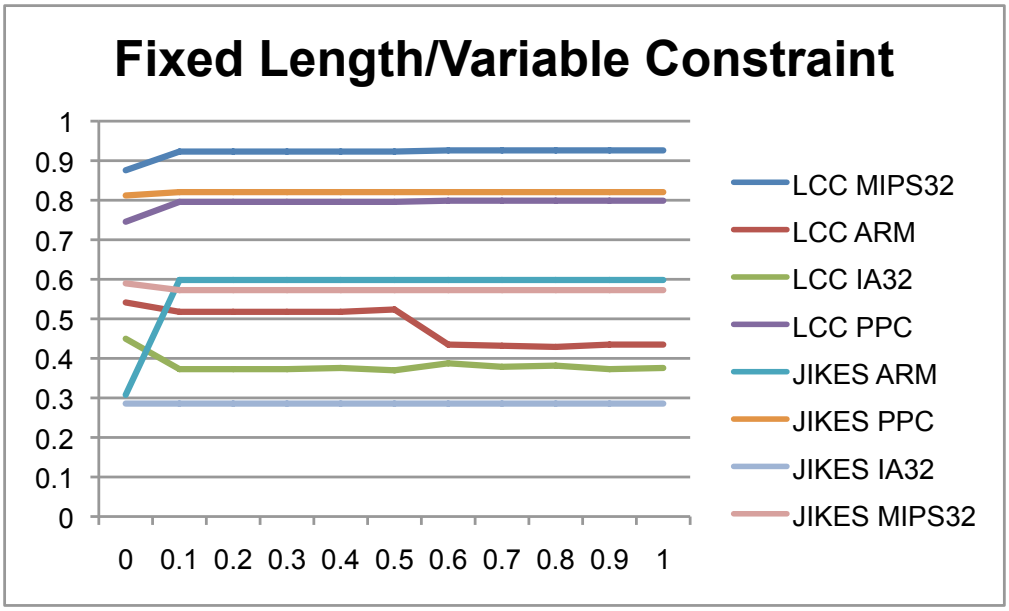


Figure 8.20: Fixed Length/Variable Constraints

CHAPTER 9

CONCLUSION

This dissertation extends the state of the art in automated compiler construction. It contributes to our understanding of instruction level semantics and how to derive instruction selector patterns automatically. It also provides insight into how search is impacted by our chosen heuristics. In this chapter, we recap our main contributions and discuss directions for future research.

9.1 Main Contributions

This dissertation describes a language and search procedure for deriving IS patterns. Our efforts led to the following main contributions: (1) the CISL machine description language, (2) parameterized compiler design, (3) admissible axioms and canonicalization, (4) solving the generalized instruction selector generator matching problem, (5) adapter programs, (6) evaluating GIST between multiple IR and ISA combinations, and (7) validating GIST against two instruction selectors (`lcc`/JikesRVM). To summarize, GIST largely solves the "UNCOL Problem" [Conway, 1958] for instruction selection. UNCOL intended to make it easy to create compilers for each new target machine and programming language. The concept of a universal IR was introduced, but was never adopted because of the wide variety of IRs that exist to support different kinds of programming language paradigms. GIST supports *any* IR and any target by describing both in the CISL language and discovering the ISA implementation for each IR, thus allowing compilers to be easily created for any IR and ISA combination.

9.1.1 CISL Instruction Specification Language

In Chapter 3, we presented the design and implementation of the CISL [Moss et al., 2005] instruction specification language, which supports a wide range of target ISAs, compiler IRs, and VM specifications while maintaining its independence from existing frameworks. We showed that it is possible and beneficial to use a universal language to describe both IR and target ISAs. CISL's compiler and target independent properties allow GIST to discover IS patterns independently of a compiler framework and target machine.

9.1.2 Compiler Design Specification

In Chapter 4, we presented the means for describing the mapping between the resources of the source and target. We do this separately from the source and target semantics because it embodies an important set of design choices over which a compiler writer must have control, e.g., how can we map a source stack abstraction onto a target memory? Keeping store mappings separate from the source and target description also avoids polluting them with semantic information that should not be directly tied to either. We also discussed CISLs support for describing complex compiler state in a fashion independent of compiler framework.

9.1.3 Admissible Axioms and Canonicalization

In Chapter 5, we described the importance of canonicalization and how CISL semantic patterns are normalized to produce canonical forms. GIST uses two techniques to normalize CISL trees: term rewriting systems and expression reassociation. Term rewriting system techniques were used to reduce the number of search-time axioms compared to previous work. We showed that canonicalization axioms are restricted to those that are terminating, confluent, and admissible—important criteria that allow IR and ISA trees to be reduced to a normal form.

9.1.4 GIST and Heuristic Beam Search

In Chapter 6 we presented GIST, a solution to the generalized instruction selector generator matching problem. We described the search algorithm we used (beam search), the GIST search strategy, how target instruction patterns are indexed, target resource allocation, the necessary application of search-time axioms, how GIST evaluates partial IS solutions using heuristics, and what those heuristics are. We demonstated the effectiveness of GIST with trace output generated during search.

9.1.5 Compiler Adapters

In Chapter 7, we presented compiler adapters that are used to adapt GIST output to an existing compiler framework. While the descriptions and search process for GIST are framework independent, at some point the generated instruction selector must interface with a code generator and front end for the results to be useful. We have worked to make creating adapters both simple and easy, and our case studies on the `lcc` C compiler and Jikes RVM Baseline bytecode compiler show that writing them does not require extensive experience with compilers, and that they are at least an order of magnitude smaller than ISA descriptions.

9.1.6 GIST Evaluation and Validation

We evaluated GIST in Chapter 8 in the context of two very different compiler frameworks and multiple target architectures, including generation of instruction selectors for targets that were not previously supported by the frameworks. We validated GIST for those targets where code generators were available. We showed that our generated instruction selectors produce code that has performance essentially identical to that of the existing selectors.

9.2 Future Directions

We made significant progress in demonstrating the effectiveness and practicality of generating instruction selectors automatically in a compiler- and target-independent fashion, but work remains to ensure that CISL and GIST can flourish with next-generation architectures (as we believe they can). This will require additional CISL descriptions, possibly more axioms to describe complex interactions in a heterogenous multi-core environment, and potentially new heuristics to guide GIST. We summarize additional future directions with CISL and GIST in the following sections.

9.2.1 CISL and GIST Extensions

We demonstrated GISTs effectiveness at generating instruction selectors automatically in a machine and compiler independent manner. At the same time, this work has demonstrated that high-level IR semantics (e.g., JVM bytecode) are still a difficult area, especially when matched against an ISA (e.g., x86) that supports upward of 100K instructions. To solve this problem requires further exploration and experimentation with the kind of search technique that is used and the choice of heuristics to apply.

There are several opportunities in this area that could lead to improved coverage for problematic cases. In particular, we believe the application of machine learning can improve coverage by specializing our heuristic function to the IR/ISA pair. Furthermore, it could be advantageous to use different search strategies at different times depending on the *complexity* of the IR and ISA. For example, GIST currently uses beam search for all combinations. Perhaps applying first a slower depth-first or breadth-first search can lead to a solution quickly, and that information can be used to guide beam search to find specialized forms. In addition, statistics from previous search runs can be used to improve the performance (i.e., quickly prune unlikely paths) and coverage of GIST.

In addition, extensions are required to support IR instructions that involve calling conventions and case analysis (e.g., JVM `tableswitch` bytecode). Instructions of this sort

require a translation into a form that can be handled by GIST and a state machine engine to record and use allocated locations, i.e. argument 3 is in a register, argument 4 is in memory at location l_1, etc. Currently, CISL can describe these translated forms and GIST can find matching ISA sequences. This has not been a problem in earlier techniques or for GIST on `lcc` combinations because most of the translation has already occurred by the front-end of the compiler to produce simpler IR operations. Additional CISL extensions are necessary to easily manage ISA subsets to allow functional simulators to be generated using a complete ISA description and instruction selectors to be generated using a subset of the instructions useful for compilers. Lastly, CISL can be used for generating additional compiler tools, such as register allocators, schedulers, assemblers, disassemblers, and linkers.

9.2.2 Peephole Optimizers

A peephole optimizer is an instruction selector that generates more optimal instructions from an ISA to itself. Although the requirements are slightly different, it would be interesting to see if we can apply GIST in this context. This would allow GIST generated instruction selector patterns to be further optimized by a GIST generated peephole optimizer.

9.2.3 Binary Rewriters

A binary rewriter translates code for one machine into code that can execute on another. Because GIST views machines in terms of CISL it would be possible to use GIST to find instruction selector patterns from one architecture to another (e.g., PowerPC to x86, MIPS to ARM). It would be interesting to see what extensions would be required for GIST to solve the binary rewriter problem.

9.2.4 Very Portable JIT Optimizer

In this dissertation we focused on the idea of CISL being used to describe target machines and compiler IR for multiple frameworks—what if we used CISL directly to represent a standard compiler IR? We could use this as the basis for constructing a compiler and ar-

chitecture independent JIT optimizer that performs transformations on CISL pattern representations. Furthermore, we could use CISL to describe compiler IRs for other frameworks and use GIST to generate rules to translate the other IR into the standard form used by our optimizer.

APPENDIX

A

CISL ISA Descriptions
ARM Excerpt

```
class Instruction {
    var inst_t inst = M.word[fetch(R[15])];
    fun stepPC(bit passed, rinx_t dest) {
        if (!passed || (passed && (dest != 15))) {
            R[15] = R[15] + 4;
        }
    }
}

mixin Eq extends Condition {
    fun encode() { cond = 0b0000; }
    fun bit condPassed() { return getZ(); }
}

class CondInstruction extends Instruction
    uses Eq, Ne, Cs, Cc, Mi, Pl, Vs, Vc,
         Hi, Ls, Ge, Lt, Gt, Le, Al, Nv {
    var big signed bit[4] cond @ inst[0:3];
}

class DataProcessingInstruction extends CondInstruction
    uses ImmediateShifterOperand, RegisterShifterOperand,
         LogicalShiftLeftImm, LogicalShiftRightImm,
         RotateRightImm,
         LogicalShiftLeftReg, LogicalShiftRightReg,
         ArithmeticShiftRightReg, RotateRightReg,
         SetCPSR, LeaveCPSR, ShifterOperandOut {
         RotateRightExtend {
    var big signed bit[4] op @ inst[7:10];
    var big signed bit[1] s  @ inst[11:11];
    var rinx_t rn @ inst[12:15];
    var rinx_t rd @ inst[16:19];
}

instruction class Mov extends DataProcessingInstruction {
    fun encode() {
        op = 0b1101;
        inst[4:5] = 0b00;
```

```
    }
    fun effect() {
        R[rd] = shifter_operand();
    }
}

instruction class Orr extends DataProcessingInstruction {
    fun encode() {
        op = 0b1100;
        inst[4:5] = 0b00;
    }
    fun effect() {
        R[rd] = R[rn] | shifter_operand();
    }
}

instruction class Ldrsb extends MiscMemInstruction {
    fun encode() {
        L = 0b1;
        S = 0b1;
        H = 0b0;
    }
    fun effect() {
        var address_t addr = get_addr();
        var byte_t val = M.byte[id(addr)];
        R[rd] = val ! 32;
    }
}
```

IA32 CISL Excerpt

```
mixin ModRM16_BX_SI_EA {
    fun encode() {
        mod = 0b00;
        r_m = 0b000;
    }
    fun word_t getLoc() {
        var addr_t offset = (BX + SI) ! 32;
        return M.word[flat(offset)];
    }
    fun setLoc(word_t word) {
        var addr_t offset = (BX + SI) ! 32;
        M.word[flat(offset)] = word;
    }
}

mixin RegEAX {
    fun encode() {
        reg = 0b000;
    }
    fun dword_t getDWordReg() {
        return EAX;
    }
    fun setDWordReg(dword_t word) {
```

```
        EAX = word;
    }
    fun word_t getWordReg() {
        return AX;
    }
    fun setWordReg(word_t word) {
        AX = word;
    }
    fun byte_t getByteReg() {
        return AL;
    }
    fun setByteReg(byte_t word) {
        AL = word;
    }
}

instruction class MOV_R2RM_16 extends OneByteOpcode
    uses RegEAX, RegECX, RegEDX, RegEBX, RegESP,
         RegEBP, RegESI, RegEDI, ModRM16_BX_SI_EA,
         ModRM16_BX_DI_EA, ModRM16_BP_SI_EA,
         ModRM16_BP_DI_EA, ModRM16_SI_EA, ModRM16_DI_EA,
         ModRM16_Disp16_EA, ModRM16_BX_EA, ModRM16_BX_SI_Disp8_EA,
         ModRM16_BX_DI_Disp8_EA, ModRM16_BP_SI_Disp8_EA,
         ModRM16_BP_DI_Disp8_EA, ModRM16_SI_Disp8_EA,
         ModRM16_DI_Disp8_EA, ModRM16_BP_Disp8_EA,
         ModRM16_BX_Disp8_EA, ModRM16_BX_SI_Disp16_EA,
         ModRM16_BX_DI_Disp16_EA, ModRM16_BP_SI_Disp16_EA,
         ModRM16_BP_DI_Disp16_EA, ModRM16_SI_Disp16_EA,
         ModRM16_DI_Disp16_EA, ModRM16_BP_Disp16_EA,
         ModRM16_BX_Disp16_EA, ModRM16_AX, ModRM16_CX,
         ModRM16_DX, ModRM16_BX, ModRM16_SP, ModRM16_BP,
         ModRM16_SI, ModRM16_DI
{
    var little unsigned bit[2] mod @ mod_rm[7:6];
    var little unsigned bit[3] reg @ mod_rm[5:3];
    var little unsigned bit[3] r_m @ mod_rm[2:0];
    fun encode() {
        D_FLAG = 0;
        inst   = 0x89;
    }
    fun effect() {
        setLoc(getWordReg());
    }
}

instruction class CMOVE_16 extends TwoByteOpcode
    uses RegEAX, RegECX, RegEDX, RegEBX, RegESP, RegEBP,
         RegESI, RegEDI, ModRM16_BX_SI_EA, ModRM16_BX_DI_EA,
         ModRM16_BP_SI_EA, ModRM16_BP_DI_EA, ModRM16_SI_EA,
         ModRM16_DI_EA, ModRM16_Disp16_EA, ModRM16_BX_EA,
         ModRM16_BX_SI_Disp8_EA, ModRM16_BX_DI_Disp8_EA,
         ModRM16_BP_SI_Disp8_EA, ModRM16_BP_DI_Disp8_EA,
         ModRM16_SI_Disp8_EA, ModRM16_DI_Disp8_EA,
         ModRM16_BP_Disp8_EA, ModRM16_BX_Disp8_EA,
```

```
          ModRM16_BX_SI_Disp16_EA, ModRM16_BX_DI_Disp16_EA,
          ModRM16_BP_SI_Disp16_EA, ModRM16_BP_DI_Disp16_EA,
          ModRM16_SI_Disp16_EA, ModRM16_DI_Disp16_EA,
          ModRM16_BP_Disp16_EA, ModRM16_BX_Disp16_EA,
          ModRM16_AX, ModRM16_CX, ModRM16_DX, ModRM16_BX,
          ModRM16_SP, ModRM16_BP, ModRM16_SI, ModRM16_DI
{
    var byte_t mod_rm = M.byte[fetchByte()];
    var little unsigned bit[2] mod @ mod_rm[7:6];
    var little unsigned bit[3] reg @ mod_rm[5:3];
    var little unsigned bit[3] r_m @ mod_rm[2:0];
    fun encode() {
        inst2 = 0x44;
        D_FLAG = 0;
    }
    fun effect() {
        if (ZF) {
            setWordReg(getLoc());
        }
    }
}
```

PowerPC CISL Excerpt

```
class Instruction {
  var inst_t inst;
  var opcd_t OPCD @ inst[0:5];
}

class AForm extends Instruction {
    var reg_t FRT @ inst[6:10];
    var reg_t FRA @ inst[11:15];
    var reg_t FRB @ inst[16:20];
    var reg_t FRC @ inst[21:25];
      var reg_t  XO @ inst[26:30];
}

class IForm extends Instruction {
    var li_t LI @ inst[6:29];
    var aa_t AA @ inst[30:30];
    var lk_t LK @ inst[31:31];
}

class DForm_RT_RA_D extends Instruction {
  var reg_t    RT @ inst[6:10];
  var reg_t    RA @ inst[11:15];
  var disp16_t D  @ inst[16:31];
}

class DForm_RT_RA_SI extends Instruction {
  var reg_t RT @ inst[6:10];
  var reg_t RA @ inst[11:15];
  var si_t  SI @ inst[16:31];
}
```

```
class DForm_BF_RA_SI extends Instruction {
  var reg_t BF @ inst[6:10];
  var reg_t RA @ inst[11:15];
  var si_t  SI @ inst[16:31];
}

class DForm_RS_RA_D extends Instruction {
  var reg_t    RS @ inst[6:10];
  var reg_t    RA @ inst[11:15];
  var disp16_t D  @ inst[16:31];
}

class DForm_FRT_RA_D extends Instruction {
  var reg_t    FRT @ inst[6:10];
  var reg_t    RA  @ inst[11:15];
  var disp16_t D   @ inst[16:31];
}

class DForm_FRS_RA_D extends Instruction {
  var reg_t    FRS @ inst[6:10];
  var reg_t    RA  @ inst[11:15];
  var disp16_t D   @ inst[16:31];
}

class DForm_TO_RA_SI extends Instruction {
  var reg_t    TO @ inst[6:10];
  var reg_t    RA  @ inst[11:15];
  var disp16_t SI   @ inst[16:31];
}

class XOForm_RT_RA_RB_XO extends Instruction {
  var reg_t    RT @ inst[6:10];
  var reg_t    RA @ inst[11:15];
  var reg_t    RB @ inst[16:20];
  var xo_t     XO @ inst[21:30];
}

class XOForm_FRT_RA_RB extends Instruction {
  var reg_t    FRT @ inst[6:10];
  var reg_t    RA @ inst[11:15];
  var reg_t    RB @ inst[16:20];
  var xo_t     XO @ inst[21:30];
}

class XForm_TO_RA_RB extends Instruction {
  var to_t     TO @ inst[6:10];
  var reg_t    RA @ inst[11:15];
  var reg_t    RB @ inst[16:20];
  var xo_t     XO @ inst[21:30];
}

class XForm_RS_RA_RB extends Instruction {
  var to_t     RS @ inst[6:10];
```

```
  var reg_t    RA @ inst[11:15];
  var reg_t    RB @ inst[16:20];
  var xo_t     XO @ inst[21:30];
}

class XForm_FRT_FRA_FRB extends Instruction {
  var to_t     FRT @ inst[6:10];
  var reg_t    FRA @ inst[11:15];
  var reg_t    FRB @ inst[16:20];
  var xo_t     XO  @ inst[21:30];
}

class XForm_FRS_RA_RB extends Instruction {
  var reg_t    FRS @ inst[6:10];
  var reg_t    RA @ inst[11:15];
  var reg_t    RB @ inst[16:20];
  var xo_t     XO @ inst[21:30];
}

class XForm_RS_RA_SH extends Instruction {
  var to_t     RS @ inst[6:10];
  var reg_t    RA @ inst[11:15];
  var reg_t    SH @ inst[16:20];
  var xo_t     XO @ inst[21:30];
}

instruction class lwz extends DForm_RT_RA_D
                      uses LoadWord, LoadWordZero {
  fun encode() {
    OPCD = 32;
  }

  fun effect() {
    R[RT] = loadWord();
  }
}

mixin LoadByteIndexed {
  fun word_t loadByte() {
    return M.byte[regIndexed(RA, RB)] !0 32;
  }
}

instruction class lbz extends DForm_RT_RA_D
                      uses LoadByte, LoadByteZero {
    fun encode() {
        OPCD = 34;
    }

    fun effect() {
        R[RT] = loadByte();
    }
}
```

```
instruction class addis extends DForm_RT_RA_SI
                        uses RegisterValueZero,
                             RegisterValueNonZero {
  fun encode() {
    OPCD = 14;
  }

  fun effect() {
    var word_t x = (type word_t) (SI!32);
    R[RT] = registerValue() + (x<<16);
  }
}

instruction class cmpi extends DForm_BF_RA_SI {
  fun encode() {
    OPCD = 11;
    BF   = 0;
  }

  fun effect() {
    EQ = (R[RA] == SI);
    NE = (R[RA] != SI);
    GT = (R[RA] >  SI);
    LT = (R[RA] <  SI);
    GE = (R[RA] >= SI);
    LE = (R[RA] <= SI);
  }
}

instruction class beq extends BForm {
  fun encode() {
    OPCD = 16;
    BI   = 0;
  }

  fun effect() {
    if (EQ) {
      PC[0] = PC[0] + (type word_t)(BD!32);
    }
  }
}
```

Sparc8 CISL Excerpt

```
class Instruction {
    var little signed bit[32]   inst;
    var little signed bit[2]    op @ inst[31:30];
}

class Format1 extends Instruction {
    var little signed bit[30]   disp30 @ inst[29:0];
}

class Format2a extends Instruction {
```

```
    var rinx_t                  rd @ inst[29:25];
    var little signed bit[3]    op2 @ inst[24:22];
    var little signed bit[22]   imm22 @ inst[21:0];

    fun encode() {
        op = 0;
    }
}

mixin IConditionCode {
    fun bit getCC_N() {
        return PSR[23];
    }
    fun bit getCC_Z() {
        return PSR[22];
    }
    fun bit getCC_V() {
        return PSR[21];
    }
    fun bit getCC_C() {
        return PSR[20];
    }
    fun setCC_N(bit val) {
        PSR[23] = val;
    }
    fun setCC_Z(bit val) {
        PSR[22] = val;
    }
    fun setCC_V(bit val) {
        PSR[21] = val;
    }
    fun setCC_C(bit val) {
        PSR[20] = val;
    }
}

instruction class BE extends Format2b uses IConditionCode {
    fun encode() {
        op2 = 2;
        cond = 1;
    }
    fun effect() {
        if (getCC_Z())
        {
            var little signed bit[32] extended = disp22!32;
            var little signed bit[32] res = 4 * extended;
            PC = PC + (type addr_t) (res);
        }
    }
}

instruction class SWAP extends Format3
            uses direct_address, offset_address{
  fun encode() { op3 = 0b001111; }
```

```
  fun effect() {
    var word_t temp = R[gen_idx(rd)];
    R[gen_idx(rd)] = Mem.word(gen_address());
    Mem.word(gen_address()) = temp;
  }
}
```

MIPS CISL Excerpt

```
instruction class DIVU extends R_Format {
    fun encode() {
        op = 0b000000;
        rd = 0;
        shamt = 0;
        funct = 0b011010;
    }
    fun effect() {
        var unsigned bit[32] urs = (type unsigned bit[32]) R[rs];
        var unsigned bit[32] urt = (type unsigned bit[32]) R[rt];
        var unsigned bit[32] lo_res = urs / urt;
        var unsigned bit[32] hi_res = urs % urt;

        var word_t s_lo_res = (type word_t) lo_res;
        var word_t s_hi_res = (type word_t) hi_res;

        LO[0] = s_lo_res;
        HI[0] = s_hi_res;
    }
}

instruction class MFLO extends R_Format {
    fun encode() {
        op = 0b000000;
        rs = 0;
        rt = 0;
        shamt = 0;
        funct = 0b010010;
    }
    fun effect() {
        R[rd] = LO[0];
    }
}

instruction class BGE extends I_Format {
    fun encode() {
        op = 0b000100;
    }

    fun effect() {
        if (R[rs] >= R[rt]) {
            PC[0] = PC[0] + (imm << 2)!32;
        }
```

```
    }
}
```

PDP-8 CISL Excerpt

```
type addr_t = big unsigned bit[12];
type val_t  = big unsigned bit[12];

type inst_t = big unsigned bit[12];
type op_t  = big unsigned bit[3];
type i_t   = big unsigned bit[1];
type z_t   = big unsigned bit[1];
type a_t   = big unsigned bit[7];

store pdp {
    indexed PC[1][addr_t] {
    }

    indexed ACC[1][val_t] {
    }

    address M[addr_t] {
        quantum data val_t val;

        mode addr_t direct(addr_t addr) {
            return addr;
        }
    }
}

class Inst uses zOn, zOff {
    var inst_t word;
    var op_t   op   @ word[0:2];
    var i_t    i    @ word[3:3];
    var z_t    z    @ word[4:4];
    var a_t    offs @ word[5:11];
}

mixin zOn {
    fun encode() {
        z = 1;
    }

    fun addr_t effAddr(a_t addr) {
        return addr!12;
    }
}

mixin zOff {
    fun encode() {
        z = 0;
    }

    fun addr_t effAddr(a_t addr) {
```

```
        return offs#!12;
    }
}

instruction class AND extends Inst {
    fun encode() {
        op = 0b000;
    }

    fun effect() {
        ACC = ACC & M.val[direct(effAddr(offs))];
    }
}

instruction class TAD extends Inst {
    fun encode() {
        op = 0b001;
    }

    fun effect() {
        ACC = ACC + M.val[direct(effAddr(offs))];
    }
}

instruction class ISZ extends Inst {
    fun encode() {
        op = 0b010;
    }

    fun effect() {
        M.val[direct(effAddr(offs))] =
          M.val[direct(effAddr(offs))] + 1;
        if (M.val[direct(effAddr(offs))] == 0) {
            PC = PC + 1;
        }
    }
}

instruction class DCA extends Inst {
    fun encode() {
        op = 0b010;
    }

    fun effect() {
        M.val[direct(effAddr(offs))] = ACC;
        ACC = 0;
    }
}

instruction class CLA extends Inst {
    fun encode() {

    }
```

```
    fun effect() {
        ACC = 0;
    }
}

instruction class SKPNE extends Inst {
    fun encode() {

    }

    fun effect() {
        if (ACC != 0) {
            PC = PC + 1;
        }
    }
}

instruction class SKPE extends Inst {
    fun encode() {

    }

    fun effect() {
        if (ACC == 0) {
            PC = PC + 1;
        }
    }
}

instruction class SETA extends Inst {
    fun encode() {

    }

    fun effect() {
        ACC = 1;
    }
}

instruction class JMP extends Inst {
    var bit[8] imm;

    fun encode() {

    }

    fun effect() {
        PC = imm!32;
    }
}

instruction class COMA extends Inst {
    fun encode() {
```

```
    }

    fun effect() {
        ACC = ~ACC;
    }
}

instruction class INCA extends Inst {
    fun encode() {

    }

    fun effect() {
        ACC = ACC + 1;
    }
}

instruction class CLRA extends Inst {
    fun encode() {

    }

    fun effect() {
        ACC = 0;
    }
}
```

CISL IR Descriptions
LCC CISL Excerpt

```
mixin MEM_DIRECT {
    fun set_byte(addr_t base, addr_t addr, byte_t item) {
        M.byte[direct(addr)] = item;
    }
    fun set_half(addr_t base, addr_t addr, half_t item) {
        M.half[direct(addr)] = item;
    }
    fun set_word(addr_t base, addr_t addr, word_t item) {
        M.word[direct(addr)] = item;
    }
    fun set_dword(addr_t base, addr_t addr, dword_t item) {
        M.dword[direct(addr)] = item;
    }
}

mixin REG_PARAM {
    var addr_t src1;
    var addr_t src2;

    fun word_t get_param1() {
        return T[src1];
    }
    fun word_t get_param2() {
```

```
        return T[src2];
    }
    fun shamt_t get_shamt() {
        return T[src1][27:31];
    }
}

instruction class ASGNU1 uses MEM_DIRECT, MEM_DISP {
    var addr_t addr;
    var addr_t base;
    var addr_t regx;

    fun effect() {
        var byte_t item = T[regx][24:31];
        set_byte(base, addr, item);
    }
}

instruction class SUBU4 uses REG_PARAM, IMM_PARAM, MEM_DISP_PARAM {
  var addr_t src1;
  var word_t const;
  var addr_t dst;

  fun effect() {
    T[dst] = get_param1() - get_param2();
  }
}
```

Jikes Explicit CISL Excerpt

```
instruction class dup_x2 extends ByteCode {
  fun encode() {
    op = 91;
  }

  fun effect() {
    //// pop the top three values from the stack
    T[0] = S.slot[direct(SP)];
    SP = SP + 1;
    T[1] = S.slot[direct(SP)];
    SP = SP + 1;
    T[2] = S.slot[direct(SP)];
    //// push the values in the required order
    S.slot[direct(SP)] = T[0];
    SP = SP - 1;
    S.slot[direct(SP)] = T[2];
    SP = SP - 1;
    S.slot[direct(SP)] = T[1];
    SP = SP - 1;
    S.slot[direct(SP)] = T[0];
  }
}
```

```
instruction class getfield_boolean extends ByteCode {
    var word_t fieldOffset;

    fun encode() {
        op = 0xb4;
    }

    fun effect() {
        T[0] = S.slot[direct(SP)];
        SP = SP - 1;
S.slot[direct(SP)] = H.byte[fieldRef(T[0],fieldOffset)]!032;
}
}

instruction class iadd extends ByteCode {
  fun encode() {
    op = 96;
  }

  fun effect() {
   S.slot[direct(SP + 1)] =
      S.slot[direct(SP)] + S.slot[direct(SP+1)];
   SP = SP + 1;
  }
}
```

Jikes Implicit CISL Excerpt

```
instruction class dup_x2 extends ByteCode {
  fun encode() {
    op = 91;
  }

  fun effect() {
    T[0] = S.slot[direct(spTopOffset)];
    T[1] = S.slot[direct(spTopOffset + 4)];
    T[2] = S.slot[direct(spTopOffset + 8)];
    S.slot[direct(spTopOffset + 8)] = T[0];
    S.slot[direct(spTopOffset + 4)] = T[2];
    S.slot[direct(spTopOffset)]     = T[1];
    S.slot[direct(spTopOffset - 4)] = T[0];
  }
}

instruction class getfield_boolean extends ByteCode {
    var word_t fieldOffset;

    fun encode() {
        op = 0xb4;
    }

    fun effect() {
        T[0] = S.slot[direct(spTopOffset)];
        S.slot[direct(spTopOffset)] =
```

```
          H.byte[fieldRef(T[0],fieldOffset)]!032;
    }
}

instruction class iadd extends ByteCode {
  fun encode() {
    op = 96;
  }

  fun effect() {
    S.slot[direct(spTopOffset+4)] =
      S.slot[direct(spTopOffset)]+S.slot[direct(spTopOffset+4)];
  }
}
```

PQCC CISL Excerpt

```
type addr_t = big unsigned bit[12];
type val_t  = big unsigned bit[12];

// The instruction word type
type inst_t = big unsigned bit[12];
type op_t  = big unsigned bit[3];
type i_t   = big unsigned bit[1];
type z_t   = big unsigned bit[1];
type a_t   = big unsigned bit[7];

store pqcc {
    indexed PC[1][addr_t] {
    }

    indexed ACC[1][val_t] {
    }

    address M[addr_t] {
        quantum data val_t val;
    }
}

instruction class setacc {
    fun effect() {
        if (ACC == 0) {
            ACC = 1;
        }
    }
}

instruction class subacc {
    var addr_t offset;

    fun effect() {
        ACC = M.val[direct(offset)] - ACC;
    }
}
```

```
instruction class load {
    var addr_t offset;

    fun effect() {
        ACC = M.val[direct(offset)];
    }
}
```

Store Schema Description Examples Jikes Implicit Baseline to PowerPC

```
map jikes to ppc {

    rule {
        pattern {
            S.slot[direct(spTopOffset)] =>
            M.word[basePlusOffset(1,spTopOffset)];
        }
    }

    rule {
        pattern {
            S.slot[direct(spTopOffset + $x)] =>
            M.word[basePlusOffset(1,spTopOffset + $x)];
        }
    }

    rule {
        pattern {
            S.slot[direct(spTopOffset - $x)] =>
            M.word[basePlusOffset(1,spTopOffset - $x)];
        }
    }

    rule {
        pattern {
            S.slot[direct(location)] =>
            M.word[basePlusOffset(1,location)];
        }
    }

    rule {
        pattern {
            S.slot[direct(location - $x)] =>
            M.word[basePlusOffset(1,location - $x)];
        }
    }

    rule {
        pattern {
            S.slot[direct(location + $x)] =>
            M.word[basePlusOffset(1,location + $x)];
        }
    }
```

```
rule {
    pattern {
        S.slot[offset(T[$x], $y)] =>
        M.word[basePlusOffset(`$x+3`,$y)];
    }
}

rule {
    pattern {
        S.slot[offset(T[$x], T[$y])] =>
        M.word[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.slot[regIndexed($x, $y)] =>
        M.word[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.dslot[direct(spTopOffset)] =>
        M.dword[basePlusOffset(1,spTopOffset)];
    }
}

rule {
    pattern {
        S.dslot[direct(spTopOffset + $x)] =>
        M.dword[basePlusOffset(1,spTopOffset + $x)];
    }
}

rule {
    pattern {
        S.dslot[direct(spTopOffset - $x)] =>
        M.dword[basePlusOffset(1,spTopOffset - $x)];
    }
}

rule {
    pattern {
        S.single[direct(spTopOffset)] =>
        M.single[basePlusOffset(1,spTopOffset)];
    }
}

rule {
    pattern {
        S.single[direct(spTopOffset + $x)] =>
        M.single[basePlusOffset(1,spTopOffset + $x)];
```

```
    }
}

rule {
    pattern {
        S.single[direct(spTopOffset - $x)] =>
        M.single[basePlusOffset(1,spTopOffset - $x)];
    }
}

rule {
    pattern {
        S.dslot[regIndexed($x,$y)] =>
        M.dword[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.dslot[direct(location + $x)] =>
        M.dword[basePlusOffset(1,location + $x)];
    }
}

rule {
    pattern {
        S.dslot[direct(location - $x)] =>
        M.dword[basePlusOffset(1,location - $x)];
    }
}

rule {
    pattern {
        S.dslot[regIndexed($x,$y)] =>
        M.dword[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.slot[localvar(index)] =>
        M.word[basePlusOffset(1,index)];
    }
}

rule {
    pattern {
        S.half[direct(spTopOffset)] =>
        M.half[basePlusOffset(1,spTopOffset)];
    }
}

rule {
    pattern {
```

```
        S.half[direct(spTopOffset + $x)] =>
        M.half[basePlusOffset(1,spTopOffset + $x)];
    }
}

rule {
    pattern {
        S.half[direct(spTopOffset - $x)] =>
        M.half[basePlusOffset(1,spTopOffset - $x)];
    }
}

rule {
    pattern {
        S.half[regIndexed($x,$y)] =>
        M.half[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.byte[regIndexed($x,$y)] =>
        M.byte[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        H.cell[constants(index)] =>
        M.word[basePlusOffset(2,index)];
    }
}

rule {
    pattern {
        H.cell[arrayRef(T[$x],$y)] =>
        M.word[basePlusOffset(`$x+3`,$y)];
    }
}

rule {
    pattern {
        H.cell[regIndexed($x, $y)] =>
        M.word[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        H.dcell[constants(offset)] =>
        M.dword[basePlusOffset(2,offset)];
    }
}
```

```
    rule {
        pattern {
            H.dcell[constants($x)] =>
            M.dword[basePlusOffset(2,$x)];
        }
    }

    rule {
        pattern {
            H.cell[fieldRef(T[$x],$y)] =>
            M.word[basePlusOffset(`$x+3`,$y)];
        }
    }

    rule {
        pattern {
            H.byte[fieldRef(T[$x],$y)] =>
            M.byte[basePlusOffset(`$x+3`,$y)];
        }
    }

    rule {
        pattern {
            H.short[fieldRef(T[$x],$y)] =>
            M.half[basePlusOffset(`$x+3`,$y)];
        }
    }

    rule {
        pattern {
            T[$x] => R[`$x+3`];
        }
    }

    rule {
        pattern {
            F[$x] => FR[`$x+1`];
        }
    }

    rule {
        pattern {
            C[$x] => CA[$x];
        }
    }

    rule {
        pattern {
            LL[$x] => R[$x];
        }
    }

}
```

Jikes Explicit Baseline to ARM

```
map jikes to arm {

    rule {
        pattern {
            H.dcell[constants(offset)] =>
            M.dword[regimm_inc(2,offset)];
        }
    }

    rule {
        pattern {
            S.dslot[direct(location - $x)] =>
            M.dword[regimm_inc(1,location - $x)];
        }
    }

    rule {
        pattern {
            S.slot[direct(location-4)] =>
            M.word[regimm_inc(13,location-4)];
        }
    }

    rule {
        pattern {
            S.slot[direct(location)] =>
            M.word[regimm_inc(13,location)];
        }
    }

    rule {
        pattern {
            S.slot[localvar(index)] =>
            M.word[regimm_inc(11,index)];
        }
    }

    rule {
        pattern {
            S.slot[offset(T[$x], $y)] =>
            M.word[regimm_inc(`$x+3`,$y)];
        }
    }

    rule {
        pattern {
            S.slot[offset(T[$x], T[$y])] =>
            M.word[regreg_inc(`$x+3`,`$y+3`)];
        }
    }
```

```
rule {
    pattern {
        S.slot[offset(T[$x], T[$y])] =>
        M.word[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.slot[regIndexed($x, $y)] =>
        M.word[regreg_inc(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.dslot[regIndexed($x,$y)] =>
        M.dword[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.dslot[regIndexed($x,$y)] =>
        M.dword[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.half[regIndexed($x,$y)] =>
        M.half[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        S.byte[regIndexed($x,$y)] =>
        M.byte[regIndexed(`$x+3`,`$y+3`)];
    }
}

rule {
    pattern {
        H.cell[constants(index)] =>
        M.word[regimm_inc(2,index)];
    }
}

rule {
    pattern {
        H.cell[fieldRef(T[$x],$y)] =>
        M.word[regimm_inc(`$x+3`,$y)];
    }
```

```
}

rule {
    pattern {
        H.byte[fieldRef(T[$x],$y)] =>
        M.byte[regimm_inc(`$x+3`,$y)];
    }
}

rule {
    pattern {
        H.short[fieldRef(T[$x],$y)] =>
        M.half[regimm_inc(`$x+3`,$y)];
    }
}

rule {
    pattern {
        S.slot[direct(SP)] =>
        M.word[id(R[13])];
    }
}

rule {
    pattern {
        S.slot[direct(SP + $x)] =>
        M.word[id(R[13] + $x)];
    }
}

rule {
    pattern {
        S.slot[direct(SP - $x)] =>
        M.word[id(R[13] - $x)];
    }
}

rule {
    pattern {
        T[$x] => R[`$x+3`];
    }
}

rule {
    pattern {
        C[0] => CPSR[1][29];
    }
}

rule {
   pattern {
       SP-1 => R[13]-4;
   }
}
```

```
    rule {
        pattern {
            SP => R[13];
        }
    }

    rule {
        pattern {
            SP+1 => R[13]+4;
        }
    }


    rule {
        pattern {
            LL[$x] => R[$x];
        }
    }

    rule {
        pattern {
            F[$x] => FR[`$x+1`];
        }
    }
}
```

LCC to x86

```
map lcc to x86linux {

    rule {
        pattern {
            M.dword[disp($reg, $off)] =>
            M.qword[flat(GPR[$reg] + $off)];
        }
    }


    rule {
        pattern {
            M.word[disp($reg, $off)] =>
            M.dword[flat(EAX + $off)];
        }
    }

    rule {
        pattern {
            M.half[disp($reg, $off)] =>
            M.word[flat(BX)];
        }
    }

    rule {
```

```
    pattern {
        M.byte[disp($reg, $off)] =>
        M.byte[flat(GPR[$reg] + $off)];
    }
}

rule {
    pattern {
        M.dword[direct(addr)] =>
        M.qword[flat(addr)];
    }
}

rule {
    pattern {
        M.word[direct(addr)] =>
        M.dword[flat(addr)];
    }
}

rule {
    pattern {
        M.half[direct(addr)] =>
        M.word[flat(addr)];
    }
}

rule {
    pattern {
        M.byte[direct(addr)] =>
        M.byte[flat(addr)];
    }
}

rule {
    pattern {
        PC[0] => EIP;
    }
}

rule {
    pattern {
        T[$x][16..31] => BX;
    }
}

rule {
    pattern {
        T[$x][24..31] => AL;
    }
}

rule {
    pattern {
```

```
        T[regx][24..31] => AL;
    }
}

rule {
    pattern {
        T[regx][16..31] => BX;
    }
}


rule {
    pattern {
        T[src] => EBX;
    }
}

rule {
    pattern {
        T[regx] => EAX;
    }
}

rule {
    pattern {
        T[src1] => EBX;
    }
}

rule {
    pattern {
        T[dst] => EAX;
    }
}

rule {
    pattern {
        T[src2] => ECX;
    }
}

rule {
    pattern {
        T[0] => EAX;
    }
}

rule {
    pattern {
        T[1] => EBX;
    }
}

rule {
```

```
        pattern {
            T[2] => ECX;
        }
    }

    rule {
        pattern {
            T[3] => EDX;
        }
    }

    rule {
        pattern {
            F[$x] => FPR[$x];
        }
    }

}
```

LCC to MIPS

```
map lcc to mips {
    rule {
        pattern {
            T[$x] => R[$x];
        }
    }

    rule {
        pattern {
            F[$x] => FPR[$x];
        }
    }

    rule {
        pattern {
            L[0] => R[31];
        }
    }

    rule {
        pattern {
            M.dword[direct($x)] =>
            M.dword[base_offset(10, $x)];
        }
    }

    rule {
        pattern {
            M.word[direct($x)] =>
            M.word[base_offset(10, $x)];
        }
    }
```

```
    rule {
        pattern {
            M.half[direct($x)] =>
            M.word[base_offset(10, $x)];
        }
    }

    rule {
        pattern {
            M.byte[direct($x)] =>
            M.byte[base_offset(10, $x)];
        }
    }

    rule {
        pattern {
            M.dword[disp($temp,$imm)] =>
            M.dword[base_offset($temp, $imm)];
        }
    }

    rule {
        pattern {
            M.word[disp($temp,$imm)] =>
            M.word[base_offset($temp, $imm)];
        }
    }

    rule {
        pattern {
            M.half[disp($temp, $imm)] =>
            M.half[base_offset($temp, $imm)];
        }
    }

    rule {
        pattern {
            M.byte[disp($temp, $imm)] =>
            M.byte[base_offset($temp, $imm)];
        }
    }
}
```

Search Time Axioms

```
axiom param_16_decomp
  LSet($l,Param($x)) =>
  LSet($l, BOr(ShRA(Param($x),IConst(16)),
            BAnd(Param($x),IConst(65535))))

axiom lbz_rule
  SExt($op,$amount) =>
  SExt(ZExt($op,$amount),$amount)
```

```
axiom lbz_rule
  SExt($op,$amount) =>
  SExt(ZExt($op,$amount),$amount)

axiom param_sext
  Neg(Param($x)) =>
  Neg(SExt(Param($x),IConst(32)))

axiom bitcomp
  BNot($x) =>
  BNot(BOr($x,$x))

axiom twos_comp
  Neg($x) =>
  Add(BNot($x),
      IConst(1))

axiom add_zero
  LSet($x,LGet($y,$z)) =>
  LSet($x,Add(IConst(0),
              LGet($y,$z)))

axiom mul_one
  LSet($x,LGet($y,$z)) =>
  LSet($x,Mul(IConst(1),
              LGet($y,$z)))

axiom mul_zero
  LSet($x,Mul(IConst(0),$y)) =>
  LSet($x,IConst(0))

axiom skip_decomp
  GAct(Eq(IConst(0),$g),$a) =>
  Seq(GAct(LNot(Eq(IConst(0),$g)),
      LSet(LGet(Mem(PC),IConst(0)),
           Add(LGet(Mem(PC),IConst(0)),
               IConst(1)))),
      GAct(true, $a))

axiom cmp_true
  GAct(Eq($x,$x),$a) =>
  GAct(true,$a)
```

Adapters

JikesRVM Adapter

```
package experiments.adapters;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.util.HashMap;
import java.util.HashSet;
```

```
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.antlr.stringtemplate.StringTemplate;

import edu.umass.cs.cogent.unicogg.selector.GistPattern;
import edu.umass.cs.cogent.unicogg.selector.Parameter;
import edu.umass.cs.cogent.unicogg.selector.ParameterConstraint;
import edu.umass.cs.cogent.unicogg.selector.SourcePattern;
import edu.umass.cs.cogent.unicogg.selector.TargetPattern;
import edu.umass.cs.cogent.unicogg.selector.TargetSequence;

/**
 * An adapter for generating JikesRVM patterns for PPC.
 *
 * @author richards
 *
 */
public class JikesPPCAdapter extends Adapter {
private final static int NUM_TEMP_REGISTERS = 6;
private final static String COMPILER_CONSTANT = "@([0-9]+)";
private Writer fileWriter;
private boolean print_whole_file;

    /**
     * Makes a JikesRVM Baseline adapter.
     */
    public JikesPPCAdapter(String filename, boolean w) {
        super("/experiments/adapters/templates/jikes/ppc/JikesPPCAsm.stg",
              filename);
        print_whole_file = w;
        try {
            File f = new File("BaselineCompilerImpl.java");
            fileWriter = new BufferedWriter(new FileWriter(f));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private Set<String> filterParams(Set<String> p) {
        Set<String> np = new HashSet<String>();
        for (String s : p) {
            if (s.equals("inst"))
                continue;
            if (s.equals("spTopOffset"))
                continue;
            StringTemplate t = templates.getInstanceOf(s);
            np.add(t.toString());
        }
        return np;
    }

    private String replaceCompilerConstants(String instructions) {
     Pattern regex = Pattern.compile(COMPILER_CONSTANT);
     Matcher regexMatcher = regex.matcher(instructions);
     int registerCount = 0;
     HashMap<String, String> tempRegisters = new HashMap<String, String>();
     // while there's still compiler constants
     while (regexMatcher.find()) {
     // if we've already seen this constant before, use the
            // same temp. register
     if (tempRegisters.containsKey(regexMatcher.group())) {
     String reg = tempRegisters.get(regexMatcher.group());
     instructions = instructions.replaceFirst(COMPILER_CONSTANT, reg);
     // if we haven't seen this before, try to use a new temp. register
```

```
 } else {
 if (registerCount > NUM_TEMP_REGISTERS) {
 throw new RegisterAllocationException();
 }
 else {
     instructions = instructions.replaceFirst(COMPILER_CONSTANT,
                                                          "T"+registerCount);
     tempRegisters.put(regexMatcher.group(), "T"+registerCount);
     registerCount++;
 }
 }
 }

 // small hack to turn "compiler constant" arithmetic
    // expressions into a java-compliant form
 instructions = instructions.replaceAll("!32", "");

 return instructions;
}

private String genPattern(GistPattern p) {
    return genEmitMethod(p.getSource(), genTargets(p));
}

private String genEmitMethod(SourcePattern source, String genTargets) {
    StringTemplate t = templates.getInstanceOf("emitMethod");
    t.setAttribute("name", source.getName());
    t.setAttribute("args", filterParams(source.getFreeParameterNames()));
    t.setAttribute("body", genTargets);
    return t.toString();
}

private String genTargets(GistPattern p) {
    String targets = "";
    List<TargetSequence> constraints = p.constraints();
    if (constraints.size() != 0) {
        targets += genIf(constraints.get(0), genTargets(p.getSource().getName(),
                                                    constraints.get(0)));
    }

    List<TargetSequence> noConstraints = p.noConstraints();
    if (noConstraints.size() == 0) {
        System.err.printf("did not find most general sequence for %s%n",
                          p.getSource().getName());
        return targets;
    }

    String general = genTargets(p.getSource().getName(), noConstraints.get(0));
    StringTemplate t = targets.equals("") ?
      templates.getInstanceOf("constraintNone")
      : templates.getInstanceOf("constraintElse");
    t.setAttribute("sequence", general);
    targets += t;

    return targets;
}

private String genIf(TargetSequence targetSequence, String genTargets) {
    String cond = "";
    for (ParameterConstraint c : targetSequence.getConstraints()) {
        if (c.isFits()) {
            if (cond.equals("")) {
                cond += String.format("asm.fits(%s, %s)", c.getParameter(),
                                      c.getValue());
            } else {
                cond += String.format("&& asm.fits(%s, %s)", c.getParameter(),
                                      c.getValue());
            }
```

```
        }
    }
    StringTemplate t = templates.getInstanceOf("constraintIf");
    t.setAttribute("cond", cond);
    t.setAttribute("sequence", genTargets);
    return t.toString();
}

private String genTargets(String sourceName, TargetSequence targetSequence) {
    String targets = "";
    for (TargetPattern p : targetSequence.getTargets()) {
        String name = p.getName();
        if (templates.isDefined(name.replace('.','-'))) {
            StringTemplate t = templates.getInstanceOf(name.replace('.', '-'));
            for (Parameter arg : p.getParameters()) {
                String val = substituteVal(arg.getValue());
                if (!val.equals("?")) {
                    try {
                        t.setAttribute(arg.getName(), val);
                    } catch (NoSuchElementException e) {
                        // do nothing
                    }
                }
            }
            targets += String.format("%s%n", t);
        } else {
            System.err.printf("error: missing template for target %s%n", name);
        }
    }

    // prologue/epilogue
    if (templates.isDefined("pe_"+sourceName.toLowerCase())) {
     StringTemplate pe = templates.getInstanceOf("pe_"+sourceName.toLowerCase());
        pe.setAttribute("inst", targets);
        targets = pe.toString() + "\n";
    }

    return replaceCompilerConstants(targets);
}

private String substituteVal(String value) {
    if (value.equals("i!>>16"))
        return "asm.maskUpper16(i)";
    if (value.equals("65535&i"))
        return "asm.maskLower16(i)";
    if (value.equals("index"))
     return "getGeneralLocalLocation(index)";
    return value;
}

/**
 * Executes the adapter.
 */
public void execute() {
    String allInstructions = "";

    //// Iterate over the selector patterns
    for (GistPattern p : patterns.getPatterns()) {
        try {
            allInstructions += genPattern(p);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            // continue
        }
    }

    System.out.println(allInstructions);
```

```
        try {
            // if we want to write the entire java file
            if (print_whole_file) {
                StringTemplate st = templates.getInstanceOf("java_stuff");
                st.setAttribute("gist",allInstructions);
                fileWriter.write(st.toString());
            } else {
                fileWriter.write(allInstructions);
            }
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * The main entry point.
     *
     * @param args
     *            program arguments
     */
    public static void main(String[] args) {
        try {
            if (args.length < 1) {
                System.out.println("usage: JikesPPCAdapter <gist solution>");
                System.exit(1);
            }

            boolean wholeFile = false;

            if (args.length > 1 && args[1].equals("java")) {
             wholeFile = true;
            }

            JikesPPCAdapter adapter = new JikesPPCAdapter(args[0], wholeFile);
            adapter.execute();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

JikesRVM PowerPC String Templates

```
group JikesPPCAsm;

emitMethod(name, args, body) ::= <<
protected final void emit_$name$($args; separator=", "$) {
  $body$
}

>>

constraintIf(cond, sequence) ::= <<
if ($cond$) {
  $sequence$
}
>>

constraintElseIf(cond, sequence) ::= <<
 else if ($cond$) {
  $sequence$
}
>>
```

```
constraintElse(sequence) ::= <<
 else {
  $sequence$
}
>>

constraintNone(sequence) ::= <<
$sequence$
>>

srcPat(name, args) ::= <<
/**
  * Emit code to implement the $name$ bytecode
  */
@Override
protected final void emit_$name$($args; separator=", "$) {

>>

/**
 * These templates map cisl instruction class names to jikes baseline
 * compiler emit code that implements the bytecode instruction.
 */
beq(bTarget)                          ::= "genCondBranch(EQ, bTarget);"
bne(bTarget)                          ::= "genCondBranch(NE, bTarget);"
bge(bTarget)                          ::= "genCondBranch(GE, bTarget);"
ble(bTarget)                          ::= "genCondBranch(LE, bTarget);"
blt(bTarget)                          ::= "genCondBranch(LT, bTarget);"
bgt(bTarget)                          ::= "genCondBranch(GT, bTarget);"
cmpd(RA,RB)                           ::= "asm.emitCMP($RA$, $RB$);"
cmpl(RA,RB)                           ::= "asm.emitCMPL($RA$, $RB$);"
cmpi(RA,SI)                           ::= "asm.emitCMPI($RA$, $SI$);"
andi(RA,RS,D)                         ::= "asm.emitANDI($RA$, $RS$, $D$);"
slw(RA,RS,RB)                         ::= "asm.emitSLW($RA$, $RS$, $RB$);"
sraw(RA,RS,RB)                        ::= "asm.emitSRAW($RA$, $RS$, $RB$);"
srawi(RA,RS,SH)                       ::= "asm.emitSRAWI($RA$, $RS$, $SH$);"
or(RA,RS,RB)                          ::= "asm.emitOR($RA$, $RS$, $RB$);"
xor(RA,RS,RB)                         ::= "asm.emitXOR($RA$, $RS$, $RB$);"
and(RA,RS,RB)                         ::= "asm.emitAND($RA$, $RS$, $RB$);"
divw(RT,RA,RB)                        ::= "asm.emitDIVW($RT$, $RA$, $RB$);"
mullw(RT,RA,RB)                       ::= "asm.emitMULLW($RT$, $RA$, $RB$);"
neg(RT,RA)                            ::= "asm.emitNEG($RT$, $RA$);"
srw(RA,RS,RB)                         ::= "asm.emitSRW($RA$, $RS$, $RB$);"
add(RT,RA,RB)                         ::= "asm.emitADD($RT$, $RA$, $RB$);"
adde(RT,RA,RB)                        ::= "asm.emitADDE($RT$, $RA$, $RB$);"
subf(RT,RA,RB)                        ::= "asm.emitSUBFC($RT$, $RA$, $RB$);"
subfze(RT,RA)                         ::= "asm.emitSUBFZE($RT$, $RA$);"
subfe(RT,RA,RB)                       ::= "asm.emitSUBFE($RT$, $RA$, $RB$);"
extsb(RA,RS)                          ::= "asm.emitEXTSB($RA$, $RS$);"
bclr_special()                        ::= "asm.emitBCLR();"
mtlr(RS)                              ::= "asm.emitMTLR($RS$);"
twlle(RA,RB)                          ::= "asm.emitTWLLE($RA$, $RB$);"
tweq0(RA)                             ::= "asm.emitTWEQ0($RA$);"
slwi(RA,RS,MB)                        ::= "asm.emitSLWI($RA$, $RS$, $MB$);"
fctiwz(FRT,FRB)                       ::= "asm.emitFCTIWZ($FRT$, $FRB$);"
LoadWord-lwz(RT,D,RA)                 ::= "asm.emitLWZ($RT$, $D$, $RA$);"
RegisterValueNonZero-addi(RA,RT,SI)   ::= "asm.emitADDI($RT$, $RA$, $SI$);"
RegisterValueZero-addi(RA,RT,SI)      ::= "asm.emitADDI($RT$, $RA$, $SI$);"
RegisterValueNonZero-addis(RA,RT,SI)  ::= "asm.emitADDIS($RT$, $RA$, $SI$);"
RegisterValueZero-addis(RA,RT,SI)     ::= "asm.emitADDIS($RT$, $RA$, $SI$);"

LoadWordIndexed-lwzx(RT,RA,RB)        ::= "asm.emitLWZX($RT$, $RA$, $RB$);"
LoadWordZero-lwz(RT,D,RA)             ::= "asm.emitLWZ($RT$, $D$, $RA$);"
LoadByte-lbz(RT,D,RA)                 ::= "asm.emitLBZ($RT$, $D$, $RA$);"
LoadByteZero-lbz(RT,D,RA)             ::= "asm.emitLBZ($RT$, $D$, $RA$);"
LoadHalf-lhz(RT,D,RA)                 ::= "asm.emitLHZ($RT$, $D$, $RA$);"
LoadHalfZero-lhz(RT,D,RA)             ::= "asm.emitLHZ($RT$, $D$, $RA$);"
LoadHalfIndexed-lhzx(RT,RA,RB)        ::= "asm.emitLHZX($RT$, $RA$, $RB$);"
```

```
LoadHalfZeroIndexed-lhzx(RT,RA,RB)   ::= "asm.emitLHZX($RT$, $RA$, $RB$);"
LoadHalfAlg-lha(RT,D,RA)             ::= "asm.emitLHA($RT$, $D$, $RA$);"
LoadHalfAlgZero-lha(RT,D,RA)         ::= "asm.emitLHA($RT$, $D$, $RA$);"
LoadHalfAlgIndexed-lhax(RT,RA,RB)    ::= "asm.emitLHAX($RT$, $RA$, $RB$);"
LoadHalfAlgIndexedZero-lhax(RT,RA,RB)::= "asm.emitLHAX($RT$, $RA$, $RB$);"
LoadDWord-lfd(FRT,D,RA)              ::= "asm.emitLFD($FRT$,$D$, $RA$);"
LoadDWordZero-lfd(FRT,D,RA)          ::= "asm.emitLFD($FRT$,$D$, $RA$);"
LoadDWordIndexed-lfdx(FRT,RA,RB)     ::= "asm.emitLFDX($FRT$, $RA$, $RB$);"
LoadDWordIndexedZero-lfdx(FRT,RA,RB) ::= "asm.emitLFDX($FRT$, $RA$, $RB$);"
LoadSWord-lfs(FRT,RA,D)              ::= "asm.emitLFS($FRT$, $D$, $RA$);"
LoadSWordZero-lfs(FRT,RA,D)              ::= "asm.emitLFS($FRT$, $D$, $RA$);"

StoreWordZero-stw(RS,D,RA)           ::= "asm.emitSTW($RS$, $D$, $RA$);"
StoreWord-stw(RS,D,RA)               ::= "asm.emitSTW($RS$, $D$, $RA$);"
StoreWordIndexed-stwx(RS,RA,RB)      ::= "asm.emitSTWX($RS$, $RA$, $RB$);"
StoreWordIndexedZero-stwx(RS,RA,RB)  ::= "asm.emitSTWX($RS$, $RA$, $RB$);"
StoreDWord-stfd(FRS,D,RA)            ::= "asm.emitSTFD($FRS$, $D$, $RA$);"
StoreDWordZero-stfd(FRS,D,RA)        ::= "asm.emitSTFD($FRS$, $D$, $RA$);"
StoreDWordIndexed-stfdx(FRS,RA,RB)   ::= "asm.emitSTFDX($FRS$, $RA$, $RB$);"
StoreDWordIndexedZero-stfdx(FRS,RA,RB)::= "asm.emitSTFDX($FRS$, $RA$, $RB$);"
StoreSWordZero-stfs(FRS,RA,D)        ::= "asm.emitSTFS($FRS$, $D$, $RA$);"
StoreSWord-stfs(FRS,RA,D)            ::= "asm.emitSTFS($FRS$, $D$, $RA$);"
StoreByteIndexed-stbx(RS,RA,RB)      ::= "asm.emitSTBX($RS$, $RA$, $RB$);"
StoreByteIndexedZero-stbx(RS,RA,RB)  ::= "asm.emitSTBX($RS$, $RA$, $RB$);"
StoreHalfIndexed-sthx(RS,RA,RB)      ::= "asm.emitSTHX($RS$, $RA$, $RB$);"
StoreHalfZeroIndexed-sthx(RS,RA,RB)  ::= "asm.emitSTHX($RS$, $RA$, $RB$);"
lwz_specialized(RT,D,RA)             ::= "asm.emitLWZ($RT$, $D$, $RA$);"
lbz_specialized(RT,D,RA)             ::= "asm.emitLBZ($RT$, $D$, $RA$);"
lhz_specialized(RT,D,RA)             ::= "asm.emitLHZ($RT$, $D$, $RA$);"
lha_specialized(RT,D,RA)             ::= "asm.emitLHA($RT$, $D$, $RA$);"
swz_specialized(RA,D)                ::= "asm.emitSWZ($RA$, $D$, $RA$);"
li(RT,RA,SI)                         ::= "asm.emitADDI($RT$, $SI$, $RA$);"
ori(RA,RS,D)                         ::= "asm.emitORI($RA$, $RS$, $D$);"

fadd(FRT,FRA,FRB)                    ::= "asm.emitFADD($FRT$, $FRA$, $FRB$);"
fadds(FRT,FRA,FRB)                   ::= "asm.emitFADDS($FRT$, $FRA$, $FRB$);"
fsub(FRT,FRA,FRB)                    ::= "asm.emitFSUB($FRT$, $FRA$, $FRB$);"
fsubs(FRT,FRA,FRB)                    ::= "asm.emitFSUBS($FRT$, $FRA$, $FRB$);"
fmul(FRT,FRA,FRB)                    ::= "asm.emitFMUL($FRT$, $FRA$, $FRB$);"
fmuls(FRT,FRA,FRB)                   ::= "asm.emitFMULS($FRT$, $FRA$, $FRB$);"
fdiv(FRT,FRA,FRB)                    ::= "asm.emitFDIV($FRT$, $FRA$, $FRB$);"
fdivs(FRT,FRA,FRB)                   ::= "asm.emitFDIVS($FRT$, $FRA$, $FRB$);"
fneg(FRT,FRB)                        ::= "asm.emitFNEG($FRT$, $FRB$);"

/**
 * The following templates are used to map cisl parameters to
 * parameters used in the jikes baseline compiler.
 */
index()             ::= "int index"
i()                 ::= "int i"
spTopOffset()       ::= ""
op()                ::= ""
inst()              ::= ""
bTarget()           ::= "int bTarget"
fieldOffset()       ::= "int fieldOffset"
val()               ::= "int val"
location()          ::= "int location"


pe_aload(inst)         ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_astore(inst)        ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_dup(inst)           ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_dup_x1(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_dup_x2(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_dup2(inst)          ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT*2;"
pe_dup2_x1(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT*2;"
pe_dup2_x2(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT*2;"
pe_iadd(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
```

```
pe_iand(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_idiv(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_imul(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_iload(inst)         ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_ior(inst)           ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_irem(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ishl(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ishr(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_istore(inst)        ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_isub(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_iushr(inst)         ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ixor(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ldc(inst)           ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_pop(inst)           ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_pop2(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_ifeq(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ifne(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ifgt(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_iflt(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ifge(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_ifle(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_if_icmpeq(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_if_icmpne(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_if_icmplt(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_if_icmpgt(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_if_icmple(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_if_icmpge(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_if_acmpeq(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_if_acmpne(inst)     ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_istore0(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_istore1(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_istore2(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_istore3(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_iconst_5(inst)      ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iconst_2(inst)      ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iconst_1(inst)      ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iconst_4(inst)      ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iconst_3(inst)      ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iconst_0(inst)      ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_aload0(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_aload1(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_aload2(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_aload3(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_aload4(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_aload5(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_astore0(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_astore1(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_astore2(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_astore3(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_astore4(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_astore5(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_iconst_m12(inst)    ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iload0(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iload1(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iload2(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iload3(inst)        ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_iconst_m1(inst)     ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_aconst_null(inst)   ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_i2l(inst)           ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"
pe_l2i(inst)           ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_iaload(inst)        ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_baload(inst)        ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_saload(inst)        ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_caload(inst)        ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_faload(inst)        ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_iastore(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*3;"
pe_fastore(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*3;"
pe_bastore(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*3;"
```

```
pe_sastore(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*3;"
pe_castore(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*3;"
pe_dastore(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*4;"
pe_lastore(inst)       ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*4;"
pe_d2i(inst)           ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_d2f(inst)           ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_f2d(inst)           ::= "$inst$spTopOffset -= BYTES_IN_STACKSLOT;"



pe_ladd(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_lsub(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_land(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_lor(inst)           ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_lxor(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_dadd(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_dsub(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_dmul(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_ddiv(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT*2;"
pe_fadd(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_fsub(inst)          ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_fmuls(inst)         ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"
pe_fdivs(inst)         ::= "$inst$spTopOffset += BYTES_IN_STACKSLOT;"

edu.umass.cs.cogent.unicogg.match.FitsConstraint(value,param) ::=
                                                  "asm.fits($param$,$i$)"

constraint_block(constraints, instructions) ::= <<
  if ($constraints; seperator=" &&") {
    $instructions$
  }

>>
```

`lcc` Adapter

```
package experiments.adapters;

import java.io.FileWriter;
import java.io.IOException;

import org.antlr.stringtemplate.StringTemplate;

import edu.umass.cs.cogent.unicogg.selector.InstructionPattern;
import edu.umass.cs.cogent.unicogg.selector.PatternSequence;
import edu.umass.cs.cogent.unicogg.selector.SelectorPattern;
import edu.umass.cs.cogent.unicogg.selector.SelectorPatternList;

public class Lccx86LinuxBursAdapter extends Adapter {

    /**
     * The template group to use with string template library.
     */
    private static final String templateResource =
      "/experiments/adapters/templates/lcc/x86/x86linux.stg";

    public Lccx86LinuxBursAdapter(String gistOutput) {
        super(templateResource, gistOutput, '<');
    }

    /**
     * The name of the burs output file.
     */
    private static final String bursFile         = "mips.md";

    /**
     * Emits the burs file.
```

```
 *
 * @param prologue
 *            the prologue of the md file
 * @param support
 *            the support rules of the md file
 * @param rules
 *            the generated rules from gist
 * @param orules
 *            the original rules from md file
 * @param epilogue
 *            the epilogue of the md file
 */
private void emitBurs(String prologue, String support, String rules,
                      String orules, String epilogue) {
    try {
        FileWriter fw = new FileWriter(bursFile);
        fw.append(prologue);
        fw.append(support);
        fw.append(rules);
        fw.append(orules);
        fw.append(epilogue);
        fw.close();
    } catch (IOException e) {
        System.err.printf("error: could not emit burs file: %s", e
                .getMessage());
    }
}

private String genBursRules(SelectorPatternList patterns) {
    StringBuilder b = new StringBuilder();
    b.append(String.format("//// Gist Generated Rules Begin%n"));
    for (SelectorPattern p : patterns) {
        try {
            b.append(genBursRule(p));
        } catch (IllegalArgumentException e) {
            //// error should have been reported---continue
        }
    }
    b.append(String.format("//// Gist Generated Rules End%n"));
    b.append(String.format("%n"));
    return b.toString();
}

private String genBursRule(SelectorPattern p) {
    return String.format("%s %s%n", genHead(p.getSourceSequence()),
                         genTail(p.getTargetSequence()));
}

private String genHead(PatternSequence s) {
    //// throw an exception if we have more than one ir pattern
    //// ---this approach expects
    //// only a single ir pattern.
    if (s.length() > 1) {
        System.err.printf("error: ir sequence if greater than 1");
        throw new IllegalArgumentException();
    }
    //// get the single ir pattern and name
    InstructionPattern p = s.get(0);

    //// grab the ir template
    String name = normalizeIrName(p.getName());
    StringTemplate t = templates.getInstanceOf(name);

    //// throw exception if no template exists
    if (t == null) {
        System.err.printf("warning: template does not exist for %s", name);
        throw new IllegalArgumentException();
    }
```

```
        return t.toString();
    }

    private String normalizeIrName(String name) {
        if (name.endsWith("_disp"))
            return name.replace("_disp", "");
        if (name.endsWith("_direct"))
            return name.replace("_direct", "");
        return name;
    }

    private String genTail(PatternSequence s) {
        StringBuilder b = new StringBuilder();
        for (InstructionPattern p : s) {
            StringTemplate t = templates.getInstanceOf(p.getName());
            b.append(t);
        }
        return b.toString();
    }

    @Override
    public void execute() {
        String prologue = templates.getInstanceOf("prologue").toString();
        String support  = templates.getInstanceOf("support").toString();
        //String rules    = genBursRules(super.patterns);
        String orules   = templates.getInstanceOf("rules").toString();
        String epilogue = templates.getInstanceOf("epilogue").toString();
        //emitBurs(prologue, support, rules, orules, epilogue);
    }

    public static void main(String[] args) {
        Adapter adapter = new Lccx86LinuxBursAdapter(args[0]);
        adapter.execute();
    }

}
```

`lcc` MIPS String Templates

```
group mips;

irPostfixes() ::= <<
_disp
_direct
_COMPR
>>

targetPostfixes() ::= <<
_oversized
>>

mixins() ::= <<
FPConditionCode
Overflow
>>

connect() ::= "; "

//// This template represents an individual burs rule
bursRule(head, tail) ::= "<head>          <tail>"

//// The following templates are the ir part of the burs rule
//// Assignments
MEM_DISP-ASGNI1()   ::= "stmt: ASGNI1(addr,reg)"
MEM_DISP-ASGNU1()   ::= "stmt: ASGNU1(addr,reg)"
MEM_DISP-ASGNI2()   ::= "stmt: ASGNI2(addr,reg)"
```

```
MEM_DISP-ASGNU2()   ::= "stmt: ASGNU2(addr,reg)"
MEM_DISP-ASGNI4()   ::= "stmt: ASGNI4(addr,reg)"
MEM_DISP-ASGNU4()   ::= "stmt: ASGNU4(addr,reg)"
MEM_DISP-ASGNP4()   ::= "stmt: ASGNP4(addr,reg)"
ASGNF4()   ::= "stmt: ASGNF4(addr,reg)"
ASGNF8()   ::= "stmt: ASGNF8(addr,reg)"

//// Indirects
INDIRI1()         ::= "reg: INDIRI1(addr)"
INDIRU1()         ::= "reg: INDIRU1(addr)"
INDIRI2()         ::= "reg: INDIRI2(addr)"
INDIRU2()         ::= "reg: INDIRU2(addr)"
INDIRI4()         ::= "reg: INDIRI4(addr)"
INDIRU4()         ::= "reg: INDIRU4(addr)"
INDIRP4()         ::= "reg: INDIRP4(addr)"
INDIRF4()      ::= "reg: INDIRF4(addr)"
INDIRF8()         ::= "reg: INDIRF8(addr)"

//// Conversion
CONVERT_1B-CVII4(cost="(a->syms[0]->u.c.v.i==1?2:LBURG_MAX)") ::= "reg: CVII4(reg)"
CONVERT_2B-CVII4(cost="(a->syms[0]->u.c.v.i==2?2:LBURG_MAX)") ::= "reg: CVII4(reg)"
CONVERT_3B-CVII4(cost="(a->syms[0]->u.c.v.i==3?2:LBURG_MAX)") ::= "reg: CVII4(reg)"
CONVERT_1B-CVUI4(cost="(a->syms[0]->u.c.v.i==1?1:LBURG_MAX)") ::= "reg: CVUI4(reg)"
CONVERT_2B-CVUI4(cost="(a->syms[0]->u.c.v.i==2?1:LBURG_MAX)") ::= "reg: CVUI4(reg)"
CONVERT_3B-CVUI4(cost="(a->syms[0]->u.c.v.i==3?1:LBURG_MAX)") ::= "reg: CVUI4(reg)"
CONVERT_1B-CVUU4(cost="(a->syms[0]->u.c.v.i==1?1:LBURG_MAX)") ::= "reg: CVUU4(reg)"
CONVERT_2B-CVUU4(cost="(a->syms[0]->u.c.v.i==2?1:LBURG_MAX)") ::= "reg: CVUU4(reg)"
CONVERT_3B-CVUU4(cost="(a->syms[0]->u.c.v.i==3?1:LBURG_MAX)") ::= "reg: CVUU4(reg)"
CVFF4()   ::= "reg: CVFF4(reg)"
CVFF8()   ::= "reg: CVFF8(reg)"
CVIF4()   ::= "reg: CVIF4(reg)"
CVIF8()   ::= "reg: CVIF8(reg)"
CVFI4_single(cost="(a->syms[0]->u.c.v.i==4?2:LBURG_MAX)") ::= "reg: CVFI4(reg)"
CVFI4_double(cost="(a->syms[0]->u.c.v.i==8?2:LBURG_MAX)") ::= "reg: CVFI4(reg)"


//// Arithmetic
DIVI4() ::= "reg: DIVI4(reg,reg)"
DIVU4() ::= "reg: DIVU4(reg,reg)"
MODI4() ::= "reg: MODI4(reg,reg)"
MODU4() ::= "reg: MODU4(reg,reg)"
MULI4() ::= "reg: MULI4(reg,reg)"
MULU4() ::= "reg: MULU4(reg,reg)"
ADDI4() ::= "reg: ADDI4(reg,rc)"
ADDP4() ::= "reg: ADDP4(reg,rc)"
ADDU4() ::= "reg: ADDU4(reg,rc)"
SUBI4() ::= "reg: SUBI4(reg,rc)"
SUBP4() ::= "reg: SUBP4(reg,rc)"
SUBU4() ::= "reg: SUBU4(reg,rc)"
NEGI4() ::= "reg: NEGI4(reg)"
REG_PARAM-REG_PARAM-ADDI4(cost="2") ::= "reg: ADDI4(reg,reg)"
IMM_PARAM-REG_PARAM-ADDI4() ::= "reg: ADDI4(reg,con)"
REG_PARAM-REG_PARAM-ADDP4(cost="2") ::= "reg: ADDP4(reg,reg)"
IMM_PARAM-REG_PARAM-ADDP4() ::= "reg: ADDP4(reg,con)"
REG_PARAM-REG_PARAM-ADDU4(cost="2") ::= "reg: ADDU4(reg,reg)"
IMM_PARAM-REG_PARAM-ADDU4() ::= "reg: ADDU4(reg,con)"
REG_PARAM-REG_PARAM-SUBI4(cost="2") ::= "reg: SUBI4(reg,reg)"
IMM_PARAM-REG_PARAM-SUBI4() ::= "reg: SUBI4(reg,con)"
REG_PARAM-REG_PARAM-SUBP4(cost="2") ::= "reg: SUBP4(reg,reg)"
IMM_PARAM-REG_PARAM-SUBP4() ::= "reg: SUBP4(reg,con)"
REG_PARAM-REG_PARAM-SUBU4(cost="2") ::= "reg: SUBU4(reg,reg)"
IMM_PARAM-REG_PARAM-SUBU4() ::= "reg: SUBU4(reg,con)"

//// Logical
BANDI4() ::= "reg: BANDI4(reg,rc)"
BANDU4() ::= "reg: BANDU4(reg,rc)"
BORI4() ::= "reg: BORI4(reg,rc)"
BORU4() ::= "reg: BORU4(reg,rc)"
```

```
BXORI4() ::= "reg: BXORI4(reg,rc)"
BXORU4() ::= "reg: BXORU4(reg,rc)"
REG_PARAM-REG_PARAM-BANDI4(cost="2") ::= "reg: BANDI4(reg,reg)"
IMM_PARAM-REG_PARAM-BANDI4() ::= "reg: BANDI4(reg,con)"
REG_PARAM-REG_PARAM-BANDU4(cost="2") ::= "reg: BANDU4(reg,reg)"
IMM_PARAM-REG_PARAM-BANDU4() ::= "reg: BANDU4(reg,con)"
REG_PARAM-REG_PARAM-BORI4(cost="2") ::= "reg: BORI4(reg,reg)"
IMM_PARAM-REG_PARAM-BORI4() ::= "reg: BORI4(reg,con)"
REG_PARAM-REG_PARAM-BORU4(cost="2") ::= "reg: BORU4(reg,reg)"
IMM_PARAM-REG_PARAM-BORU4() ::= "reg: BORU4(reg,con)"
REG_PARAM-REG_PARAM-BXORI4(cost="2") ::= "reg: BXORI4(reg,reg)"
IMM_PARAM-REG_PARAM-BXORI4() ::= "reg: BXORI4(reg,con)"
REG_PARAM-REG_PARAM-BXORU4(cost="2") ::= "reg: BXORU4(reg,reg)"
IMM_PARAM-REG_PARAM-BXORU4() ::= "reg: BXORU4(reg,con)"
BCOMI4() ::= "reg: BCOMI4(reg)"
BCOMU4() ::= "reg: BCOMU4(reg)"

//// Shift
IMM_PARAM-LSHI4() ::= "reg: LSHI4(reg,con5)"
IMM_PARAM-LSHU4() ::= "reg: LSHU4(reg,con5)"
IMM_PARAM-RSHI4() ::= "reg: RSHI4(reg,con5)"
IMM_PARAM-RSHU4() ::= "reg: RSHU4(reg,con5)"
REG_PARAM-LSHI4(cost="2") ::= "reg: LSHI4(reg,reg)"
REG_PARAM-LSHU4(cost="2") ::= "reg: LSHU4(reg,reg)"
REG_PARAM-RSHI4(cost="2") ::= "reg: RSHI4(reg,reg)"
REG_PARAM-RSHU4(cost="2") ::= "reg: RSHU4(reg,reg)"

//// Load
LOADI1(cost="move(a)") ::= "reg: LOADI1(reg)"
LOADU1(cost="move(a)") ::= "reg: LOADU1(reg)"
LOADI2(cost="move(a)") ::= "reg: LOADI2(reg)"
LOADU2(cost="move(a)") ::= "reg: LOADU2(reg)"
LOADI4(cost="move(a)") ::= "reg: LOADI4(reg)"
LOADU4(cost="move(a)") ::= "reg: LOADU4(reg)"
LOADP4(cost="move(a)") ::= "reg: LOADP4(reg)"
LOADF4(cost="move(a)") ::= "reg: LOADF4(reg)"
LOADF8(cost="move(a)") ::= "reg: LOADF8(reg)"

//// Comparison/Branching
JUMPV() ::= "stmt: JUMPV(acon)"
EQI4() ::= "stmt: EQI4(reg,reg)"
EQU4() ::= "stmt: EQU4(reg,reg)"
GEI4() ::= "stmt: GEI4(reg,reg)"
GEU4() ::= "stmt: GEU4(reg,reg)"
GTI4() ::= "stmt: GTI4(reg,reg)"
GTU4() ::= "stmt: GTU4(reg,reg)"
LEI4() ::= "stmt: LEI4(reg,reg)"
LEU4() ::= "stmt: LEU4(reg,reg)"
LTI4() ::= "stmt: LTI4(reg,reg)"
LTU4() ::= "stmt: LTU4(reg,reg)"
NEI4() ::= "stmt: NEI4(reg,reg)"
NEU4() ::= "stmt: NEU4(reg,reg)"
CALLI4() ::= "reg: CALLI4(ar)"
CALLP4() ::= "reg: CALLP4(ar)"
CALLU4() ::= "reg: CALLU4(ar)"
CALLF4() ::= "reg: CALLF4(ar)"
CALLF8() ::= "reg: CALLF8(ar)"
CALLV() ::= "stmt: CALLV(ar)"

//// Floating-Point Arithmetic
ADDF4() ::= "reg: ADDF4(reg,reg)"
ADDF8() ::= "reg: ADDF8(reg,reg)"
SUBF4() ::= "reg: SUBF4(reg,reg)"
SUBF8() ::= "reg: SUBF8(reg,reg)"
MULF4() ::= "reg: MULF4(reg,reg)"
MULF8() ::= "reg: MULF8(reg,reg)"
DIVF4() ::= "reg: DIVF4(reg,reg)"
DIVF8() ::= "reg: DIVF8(reg,reg)"
```

```
NEGF4() ::= "reg: NEGF4(reg)"
NEGF8() ::= "reg: NEGF8(reg)"

//// Floating-Point Comparison/Branching
EQF4() ::= "stmt: EQF4(reg,reg)"
EQF8() ::= "stmt: EQF8(reg,reg)"
GEF4() ::= "stmt: GEF4(reg,reg)"
GEF8() ::= "stmt: GEF8(reg,reg)"
GTF4() ::= "stmt: GTF4(reg,reg)"
GTF8() ::= "stmt: GTF8(reg,reg)"
LEF4() ::= "stmt: LEF4(reg,reg)"
LEF8() ::= "stmt: LEF8(reg,reg)"
LTF4() ::= "stmt: LTF4(reg,reg)"
LTF8() ::= "stmt: LTF8(reg,reg)"
NEF4() ::= "stmt: NEF4(reg,reg)"
NEF8() ::= "stmt: NEF8(reg,reg)"

//// MIPS-Specific Nested Intermediates
INDI1CVI4() ::= "reg:  CVII4(INDIRI1(addr))"
INDI2CVI4() ::= "reg:  CVII4(INDIRI2(addr))"
INDU1CVU4() ::= "reg:  CVUU4(INDIRU1(addr))"
INDU2CVU4() ::= "reg:  CVUU4(INDIRU2(addr))"
INDU1CVI4() ::= "reg:  CVUI4(INDIRU1(addr))"
INDU2CVI4() ::= "reg:  CVUI4(INDIRU2(addr))"

//// These templates are for the target instructions
SB(rs,imm,rt)           ::= <<"sb $%1,%0\n"  1>>
SH(rs,imm,rt)           ::= <<"sh $%1,%0\n"   1>>
SW(rt,imm,rs)           ::= <<"sw $%1,%0\n"   1>>
SWC1(rt,imm,rs) ::= <<"s.s $f%1,%0\n"  1>>
SDC1(rt,imm,rs) ::= <<"s.d $f%1,%0\n"  1>>
LB(rt,imm,rs)           ::= <<"lb $%c,%0\n"  1>>
LBU(rt,imm,rs) ::= <<"lbu $%c,%0\n"  1>>
LH(rt,imm,rs)           ::= <<"lh $%c,%0\n"  1>>
LHU(rt,imm,rs) ::= <<"lhu $%c,%0\n"  1>>
LW(rt,imm,rs)           ::= <<"lw $%c,%0\n"   1>>
LWC1(rt,imm,rs) ::= <<"l.s $f%c,%0\n"  1>>
LDC1(rt,imm,rs) ::= <<"l.d $f%c,%0\n"  1>>
DIV_PSEUDO(rd,rs,rt) ::= <<"div $%c,$%0,$%1\n"   1>>
DIVU_PSEUDO(rd,rs,rt) ::= <<"divu $%c,$%0,$%1\n"   1>>
REM(rd,rs,rt) ::= <<"rem $%c,$%0,$%1\n"   1>>
REMU(rd,rs,rt) ::= <<"remu $%c,$%0,$%1\n"  1>>
MUL(rd,rs,rt) ::= <<"mul $%c,$%0,$%1\n"   1>>
MULTU(rs,rt) ::= <<"multu $%0,$%1\n"  1>>
ADD_R(rd,rs,rt) ::= <<"addu $%c,$%0,$%1\n"  1>>
ADDI(rd,rs,imm="%1") ::= <<"addiu $%c,$%0,<imm>\n"  1>>
ADDU_R(rd,rs,rt) ::= <<"addu $%c,$%0,$%1\n"  1>>
ADDIU(rd,rs,imm="%1") ::= <<"addiu $%c,$%0,<imm>\n"  1>>
SUB_R(rd,rs,rt) ::= <<"subu $%c,$%0,$%1\n"  1>>
SUB_C(rd,rs,imm="%1") ::= <<"subu $%c,$%0,<imm>\n"  1>>
SUBU_R(rd,rs,rt) ::= <<"subu $%c,$%0,$%1\n"  1>>
SUBU_C(rd,rs,imm="%1") ::= <<"subu $%c,$%0,<imm>\n"  1>>
SUBU_SPECIALIZED(rt,rs) ::= <<"negu $%c,$%0\n"  1>>
AND_R(rd,rs,rt) ::= <<"and $%c,$%0,$%1\n"   1>>
ANDI(rd,rs,imm="%1") ::= <<"and $%c,$%0,<imm>\n"   1>>
OR_R(rd,rs,rt) ::= <<"or $%c,$%0,$%1\n"   1>>
ORI(rd,rs,imm="%1") ::= <<"or $%c,$%0,<imm>\n"   1>>
XOR_R(rd,rs,rt) ::= <<"xor $%c,$%0,$%1\n"   1>>
XORI(rd,rs,imm="%1") ::= <<"xor $%c,$%0,<imm>\n"   1>>
NOT(rt,rs) ::= <<"not $%c,$%0\n"   1>>
SLLV(rd,rs,rt) ::= <<"sllv $%c,$%0,$%1\n"  1>>
SRAV(rd,rs,rt) ::= <<"srav $%c,$%0,$%1\n"  1>>
SRLV(rd,rs,rt) ::= <<"srlv $%c,$%0,$%1\n"  1>>
SLL(rt,rs,imm="%1") ::= <<"sll $%c,$%0,<imm>\n"  1>>
SRL(rt,rs,imm="%1") ::= <<"srl $%c,$%0,<imm>\n"  1>>
SRA(rt,rs,imm="%1") ::= <<"sra $%c,$%0,<imm>\n"  1>>
MOVE(rt,rs) ::= <<"move $%c,$%0\n"  0>>
B(imm) ::= <<"b %0\n"   1>>
```

```
BEQ(rs,rt,imm) ::= <<"beq $%0,$%1,%a\n"   1>>
BGE(rs,rt,imm) ::= <<"bge $%0,$%1,%a\n"   1>>
BGEU(rs,rt,imm) ::= <<"bgeu $%0,$%1,%a\n"   1>>
BGT(rs,rt,imm) ::= <<"bgt $%0,$%1,%a\n"   1>>
BGTU(rs,rt,imm) ::= <<"bgtu $%0,$%1,%a\n"   1>>
BLE(rs,rt,imm) ::= <<"ble $%0,$%1,%a\n"   1>>
BLEU(rs,rt,imm) ::= <<"bleu $%0,$%1,%a\n"   1>>
BLT(rs,rt,imm) ::= <<"blt $%0,$%1,%a\n"   1>>
BLTU(rs,rt,imm) ::= <<"bltu $%0,$%1,%a\n"   1>>
BEQ(rs="$%0",rt="$%1",imm="%a") ::= <<"beq <rs>,<rt>,<imm>\n"   1>>
BNE(rs,rt,imm) ::= <<"bne $%0,$%1,%a\n"   1>>
JAL(imm) ::= <<"jal %0\n"  1>>
MFLO(rd) ::= <<"mflo $%c\n"  1>>
ADD_S(rd,rs,rt) ::= <<"add.s $f%c,$f%0,$f%1\n"  1>>
ADD_D(rd,rs,rt) ::= <<"add.d $f%c,$f%0,$f%1\n"  1>>
SUB_S(rd,rs,rt) ::= <<"sub.s $f%c,$f%0,$f%1\n"  1>>
SUB_D(rd,rs,rt) ::= <<"sub.d $f%c,$f%0,$f%1\n"  1>>
MUL_S(rd,rs,rt) ::= <<"mul.s $f%c,$f%0,$f%1\n"  1>>
MUL_D(rd,rs,rt) ::= <<"mul.d $f%c,$f%0,$f%1\n"  1>>
DIV_S(rd,rs,rt) ::= <<"div.s $f%c,$f%0,$f%1\n"  1>>
DIV_D(rd,rs,rt) ::= <<"div.d $f%c,$f%0,$f%1\n"  1>>
NEG_S(rd,rs) ::= <<"neg.s $f%c,$f%0\n"       1>>
NEG_D(rd,rs) ::= <<"neg.d $f%c,$f%0\n"       1>>
MOV_S(rt,rs) ::= <<"mov.s $f%c,$f%0\n"       0>>
MOV_D(rt,rs) ::= <<"mov.d $f%c,$f%0\n"       0>>
BC1T(offset) ::= <<"bc1t %a\n"  1>>
BC1F(offset) ::= <<"bc1f %a\n"  1>>
C_eq_S(rs,rt) ::= <<"c.eq.s $f%0,$f%1\n"  1>>
C_eq_D(rs,rt) ::= <<"c.eq.d $f%0,$f%1\n"  1>>
C_ngt_S(rs,rt) ::= <<"c.ule.s $f%0,$f%1\n"  1>>
C_ngt_D(rs,rt) ::= <<"c.ule.d $f%0,$f%1\n"  1>>
C_nge_S(rs,rt) ::= <<"c.ult.s $f%0,$f%1\n"  1>>
C_nge_D(rs,rt) ::= <<"c.ult.d $f%0,$f%1\n"  1>>
C_lt_S(rs,rt) ::= <<"c.lt.s $f%0,$f%1\n"  1>>
C_lt_D(rs,rt) ::= <<"c.lt.d $f%0,$f%1\n"  1>>
C_le_S(rs,rt) ::= <<"c.le.s $f%0,$f%1\n"  1>>
C_le_D(rs,rt) ::= <<"c.le.d $f%0,$f%1\n"  1>>
MTC1(rs,rt) ::= <<"mtc1 $%0,$f%c\n"  1>>
MFC1(rs,rt) ::= <<"mfc1 $%c,$f2\n"  1>>
CVT_S_W(rs,rt) ::= <<"cvt.s.w $f%c,$f%c\n"  1>>
CVT_S_D(rs,rt) ::= <<"cvt.s.d $f%c,$f%0\n"  1>>
CVT_D_W(rs,rt) ::= <<"cvt.d.w $f%c,$f%c\n"  1>>
CVT_D_S(rs,rt) ::= <<"cvt.d.s $f%c,$f%0\n"  1>>
TRUNC_W_S(rs,rt) ::= <<"trunc.w.s $f2,$f%0,$%c\n" 1>>
TRUNC_W_D(rs,rt) ::= <<"trunc.w.d $f2,$f%0,$%c\n" 1>>
```

`lcc` IA32 String Templates

```
group x86linux;

targetPostfixes() ::= <<
_R2RM
_RM2R
_IMM2RM
_RM
>>

mixins() ::= <<
ModRM16_Disp16_EA
ModRM32_EAX
ModRM32_EAX_Disp32_EA
ModRM32_EBX
ModRM32_EBX_EA
ModRM32_ECX
ModRM32_EDI
ModRM32_EBP
ModRM32_Disp32_EA
```

```
RegEAX
RegEBX
RegEBP
RegECX
RegEDI
RegESI
>>

connect() ::= "\\n"

bursRule(head, tail) ::= "<head>          <tail>"

//// Assignment
ASGNI1() ::= "stmt: ASGNI1(addr,rc)"
ASGNI2() ::= "stmt: ASGNI2(addr,rc)"
ASGNI4() ::= "stmt: ASGNI4(addr,rc)"
ASGNU1() ::= "stmt: ASGNU1(addr,rc)"
ASGNU2() ::= "stmt: ASGNU2(addr,rc)"
ASGNU4() ::= "stmt: ASGNU4(addr,rc)"
ASGNP4() ::= "stmt: ASGNP4(addr,rc)"

//// Arithmetic
REG_PARAM-REG_PARAM-ADDI4() ::= "reg: ADDI4(reg,mrc)"
REG_PARAM-REG_PARAM-ADDP4() ::= "reg: ADDP4(reg,mrc)"
REG_PARAM-REG_PARAM-ADDU4() ::= "reg: ADDU4(reg,mrc)"
INCI(cost="memop(a)") ::= "stmt: ASGNI4(addr,ADDI4(mem4,con1))"
INCU(cost="memop(a)") ::= "stmt: ASGNI4(addr,ADDU4(mem4,con1))"
INCP(cost="memop(a)") ::= "stmt: ASGNP4(addr,ADDP4(mem4,con1))"
DECI(cost="memop(a)") ::= "stmt: ASGNI4(addr,SUBI4(mem4,con1))"
DECU(cost="memop(a)") ::= "stmt: ASGNI4(addr,SUBU4(mem4,con1))"
DECP(cost="memop(a)") ::= "stmt: ASGNP4(addr,SUBP4(mem4,con1))"
ADDI_M(cost="memop(a)") ::= "stmt: ASGNI4(addr,ADDI4(mem4,rc))"
ADDU_M(cost="memop(a)") ::= "stmt: ASGNU4(addr,ADDU4(mem4,rc))"
SUBI_M(cost="memop(a)") ::= "stmt: ASGNI4(addr,SUBI4(mem4,rc))"
SUBU_M(cost="memop(a)") ::= "stmt: ASGNU4(addr,SUBU4(mem4,rc))"
ANDI_M(cost="memop(a)") ::= "stmt: ASGNI4(addr,BANDI4(mem4,rc))"
ANDU_M(cost="memop(a)") ::= "stmt: ASGNU4(addr,BANDU4(mem4,rc))"
ORI_M(cost="memop(a)") ::= "stmt: ASGNI4(addr,BORI4(mem4,rc))"
ORU_M(cost="memop(a)") ::= "stmt: ASGNU4(addr,BORU4(mem4,rc))"
XORI_M(cost="memop(a)") ::= "stmt: ASGNI4(addr,BXORI4(mem4,rc))"
XORU_M(cost="memop(a)") ::= "stmt: ASGNU4(addr,BXORU4(mem4,rc))"
NOTI_M(cost="memop(a)") ::= "stmt: ASGNI4(addr,BCOMI4(mem4))"
NOTU_M(cost="memop(a)") ::= "stmt: ASGNU4(addr,BCOMU4(mem4))"
NEG_M(cost="memop(a)") ::= "stmt: ASGNI4(addr,NEGI4(mem4))"
REG_PARAM-REG_PARAM-SUBI4() ::= "reg: SUBI4(reg,mrc)"
REG_PARAM-REG_PARAM-SUBP4() ::= "reg: SUBP4(reg,mrc)"
REG_PARAM-REG_PARAM-SUBU4() ::= "reg: SUBU4(reg,mrc)"
NEGI4() ::= "reg: NEGI4(reg)"
MULI4() ::= ""
MULU4() ::= "reg: MULU4(reg,mr)"
DIVI4() ::= "reg: DIVI4(reg,reg)"
DIVU4() ::= "reg: DIVU4(reg,reg)"
MODI4() ::= "reg: MODI4(reg,reg)"
MODU4() ::= "reg: MODU4(reg,reg)"
NEGI4() ::= "reg: NEGI4(reg)"

//// Logical
REG_PARAM-REG_PARAM-BANDI4() ::= "reg: BANDI4(reg,reg)"
REG_PARAM-REG_PARAM-BANDU4() ::= "reg: BANDU4(reg,reg)"
REG_PARAM-REG_PARAM-BORI4() ::= "reg: BORI4(reg,reg)"
REG_PARAM-REG_PARAM-BORU4() ::= "reg: BORU4(reg,reg)"
REG_PARAM-REG_PARAM-BXORI4() ::= "reg: BXORI4(reg,reg)"
REG_PARAM-REG_PARAM-BXORU4() ::= "reg: BXORU4(reg,reg)"
IMM_PARAM-REG_PARAM-BANDI4() ::= "reg: BANDI4(reg,con)"
IMM_PARAM-REG_PARAM-BANDU4() ::= "reg: BANDU4(reg,con)"
IMM_PARAM-REG_PARAM-BORI4() ::= "reg: BORI4(reg,con)"
IMM_PARAM-REG_PARAM-BORU4() ::= "reg: BORU4(reg,con)"
IMM_PARAM-REG_PARAM-BXORI4() ::= "reg: BXORI4(reg,con)"
```

```
IMM_PARAM-REG_PARAM-BXORU4() ::= "reg: BXORU4(reg,con)"
BCOMI4() ::= "reg: BCOMI4(reg)"
BCOMU4() ::= "reg: BCOMU4(reg)"

//// Shift
REG_PARAM-REG_PARAM-LSHI4() ::= "reg: LSHI4(reg,reg5)"
REG_PARAM-REG_PARAM-LSHU4() ::= "reg: LSHU4(reg,reg5)"
REG_PARAM-REG_PARAM-RSHI4() ::= "reg: RSHI4(reg,reg5)"
REG_PARAM-REG_PARAM-RSHU4() ::= "reg: RSHU4(reg,reg5)"
IMM_PARAM-REG_PARAM-LSHI4() ::= "reg: LSHI4(reg,con5)"
IMM_PARAM-REG_PARAM-LSHU4() ::= "reg: LSHU4(reg,con5)"
IMM_PARAM-REG_PARAM-RSHI4() ::= "reg: RSHI4(reg,con5)"
IMM_PARAM-REG_PARAM-RSHU4() ::= "reg: RSHU4(reg,con5)"

//// Comparison/Branching
REG_PARAM-MEM_DISP_PARAM-EQI4(cost="5") ::= "stmt: EQI4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-GEI4(cost="5") ::= "stmt: GEI4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-GEU4(cost="5") ::= "stmt: GEU4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-GTI4(cost="5") ::= "stmt: GTI4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-GTU4(cost="5") ::= "stmt: GTU4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-LEI4(cost="5") ::= "stmt: LEI4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-LEU4(cost="5") ::= "stmt: LEU4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-LTI4(cost="5") ::= "stmt: LTI4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-LTU4(cost="5") ::= "stmt: LTU4(mem4,reg)"
REG_PARAM-MEM_DISP_PARAM-NEI4(cost="5") ::= "stmt: NEI4(mem4,reg)"
IMM_PARAM-MEM_DISP_PARAM-EQI4(cost="5") ::= "stmt: EQI4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-GEI4(cost="5") ::= "stmt: GEI4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-GEU4(cost="5") ::= "stmt: GEU4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-GTI4(cost="5") ::= "stmt: GTI4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-GTU4(cost="5") ::= "stmt: GTU4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-LEI4(cost="5") ::= "stmt: LEI4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-LEU4(cost="5") ::= "stmt: LEU4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-LTI4(cost="5") ::= "stmt: LTI4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-LTU4(cost="5") ::= "stmt: LTU4(mem4,con)"
IMM_PARAM-MEM_DISP_PARAM-NEI4(cost="5") ::= "stmt: NEI4(mem4,con)"
MEM_DISP_PARAM-REG_PARAM-EQI4(cost="4") ::= "stmt: EQI4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-EQU4(cost="4") ::= "stmt: EQU4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-GEI4(cost="4") ::= "stmt: GEI4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-GEU4(cost="4") ::= "stmt: GEU4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-GTI4(cost="4") ::= "stmt: GTI4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-GTU4(cost="4") ::= "stmt: GTU4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-LEI4(cost="4") ::= "stmt: LEI4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-LEU4(cost="4") ::= "stmt: LEU4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-LTI4(cost="4") ::= "stmt: LTI4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-LTU4(cost="4") ::= "stmt: LTU4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-NEI4(cost="4") ::= "stmt: NEI4(reg,memx)"
MEM_DISP_PARAM-REG_PARAM-NEU4(cost="4") ::= "stmt: NEU4(reg,memx)"
REG_PARAM-REG_PARAM-EQI4(cost="4") ::= "stmt: EQI4(reg,reg)"
REG_PARAM-REG_PARAM-EQU4(cost="4") ::= "stmt: EQU4(reg,reg)"
REG_PARAM-REG_PARAM-GEI4(cost="4") ::= "stmt: GEI4(reg,reg)"
REG_PARAM-REG_PARAM-GEU4(cost="4") ::= "stmt: GEU4(reg,reg)"
REG_PARAM-REG_PARAM-GTI4(cost="4") ::= "stmt: GTI4(reg,reg)"
REG_PARAM-REG_PARAM-GTU4(cost="4") ::= "stmt: GTU4(reg,reg)"
REG_PARAM-REG_PARAM-LEI4(cost="4") ::= "stmt: LEI4(reg,reg)"
REG_PARAM-REG_PARAM-LEU4(cost="4") ::= "stmt: LEU4(reg,reg)"
REG_PARAM-REG_PARAM-LTI4(cost="4") ::= "stmt: LTI4(reg,reg)"
REG_PARAM-REG_PARAM-LTU4(cost="4") ::= "stmt: LTU4(reg,reg)"
REG_PARAM-REG_PARAM-NEI4(cost="4") ::= "stmt: NEI4(reg,reg)"
REG_PARAM-REG_PARAM-NEU4(cost="4") ::= "stmt: NEU4(reg,reg)"
IMM_PARAM-REG_PARAM-EQI4(cost="4") ::= "stmt: EQI4(reg,con)"
IMM_PARAM-REG_PARAM-EQU4(cost="4") ::= "stmt: EQU4(reg,con)"
IMM_PARAM-REG_PARAM-GEI4(cost="4") ::= "stmt: GEI4(reg,con)"
IMM_PARAM-REG_PARAM-GEU4(cost="4") ::= "stmt: GEU4(reg,con)"
IMM_PARAM-REG_PARAM-GTI4(cost="4") ::= "stmt: GTI4(reg,con)"
IMM_PARAM-REG_PARAM-GTU4(cost="4") ::= "stmt: GTU4(reg,con)"
IMM_PARAM-REG_PARAM-LEI4(cost="4") ::= "stmt: LEI4(reg,con)"
IMM_PARAM-REG_PARAM-LEU4(cost="4") ::= "stmt: LEU4(reg,con)"
IMM_PARAM-REG_PARAM-LTI4(cost="4") ::= "stmt: LTI4(reg,con)"
```

```
IMM_PARAM-REG_PARAM-LTU4(cost="4") ::= "stmt: LTU4(reg,con)"
IMM_PARAM-REG_PARAM-NEI4(cost="4") ::= "stmt: NEI4(reg,con)"
IMM_PARAM-REG_PARAM-NEU4(cost="4") ::= "stmt: NEU4(reg,con)"
CALLI4() ::= "reg:  CALLI4(addrj)"
CALLP4() ::= "reg:  CALLP4(addrj)"
CALLU4() ::= "reg:  CALLU4(addrj)"
CALLF4() ::= "freg: CALLF4(addrj)"
CALLV() ::= "stmt: CALLV(addrj)"
JUMPV() ::= "stmt: JUMPV(addrj)"

//// Templates for target instructions
ADD_32() ::= <<"?movl %0,%c\naddl %1,%c\n"  1>>
AND_32() ::= <<"?movl %0,%c\nandl %1,%c\n"  1>>
CMP_32() ::= <<"cmpl %1,%0\n"  0>>
CMOVC_32() ::= <<"cmovc" 0>>
CMOVE_32() ::= <<"cmove" 0>>
DEC_32() ::= <<"decl %1\n"  0>>
JE_32() ::= <<"je %a\n"  0>>
JGE_32() ::= <<"jge %a\n"  0>>
JG_32() ::= <<"jg %a\n"  0>>
JLE_32() ::= <<"jle %a\n"  0>>
JL_32() ::= <<"jl %a\n"  0>>
JNE_32() ::= <<"jne %a\n"  0>>
JMP_abs16() ::= <<"call %0\n" 1>>
JMP_rel16() ::= <<"jmp %0\n"  3>>
MOV_16() ::= <<"movw %1,%0\n"   1>>
MOV_32() ::= <<"movl %1,%0\n"   1>>
NEG_32() ::= <<"?movl %0,%c\nnegl %c\n"  2>>
NOT_32() ::= <<"?movl %0,%c\nnotl %c\n"  2>>
OR_32() ::= <<"?movl %0,%c\norl %1,%c\n"   1>>
SAL_32() ::= <<"?movl %0,%c\nsall %1,%c\n"  2>>
SAR_32() ::= <<"?movl %0,%c\nsarl %1,%c\n"  2>>
SHL_32() ::= <<"?movl %0,%c\nshll %1,%c\n"  2>>
SHR_32() ::= <<"?movl %0,%c\nshrl %1,%c\n"  2>>
SUB_32() ::= <<"?movl %0,%c\nsubl %1,%c\n"  1>>
XOR_32() ::= <<"?movl %0,%c\nxorl %1,%c\n"  1>>
```

BIBLIOGRAPHY

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4): 491–516, 1989. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/69558.75700.

Philippe Aigrain, Susan L. Graham, Robert R. Henry, Marshall Kirk McKusick, and Eduardo Pelegri-Llopart. Experience with a Graham-Glanville Style Code Generator. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler Construction*, pages 13–24, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3. doi: http://doi.acm.org/10.1145/502874.502876.

B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño—a Compiler-supported Java Virtual Machine for Servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999a. ACM.

B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Syst. J.*, 39 (1):211–238, January 2000a. ISSN 0018-8670. doi: 10.1145/320385.320418. URL `http://dx.doi.org/10.1145/320385.320418`.

B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *ISJ*, 44:399–417, 2005.

Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen E. Smith. Implementing Jalapeño in Java. In *OOPSLA*, pages 314–324, 1999b.

Bowen Alpern, Dick Attanasio, John J. Barton, M. G. Burke, Perry Cheng, J.-D. Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, Mark Mergen, Ton Ngo, J. R. Russell, Vivek Sarkar, Manuel J. Serrano, Janice Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *ISJ*, 39(1), February 2000b. URL `http://www.research.ibm.com/journal/sj/391/alpern.pdf`.

Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-58388-8.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 394–403, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-451-0. doi: http://doi.acm.org/10.1145/1168857.1168906.

Gordon C. Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, 1971.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

Frederick P. Brooks, Jr. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-83595-9.

R. G. G. Cattell. Automatic Derivation of Code Generators from Machine Descriptions. *ACM Trans. Program. Lang. Syst.*, 2(2):173–190, 1980. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357094.357097.

Roderic Geoffrey Galton Cattell. *Formalization and automatic derivation of code generators*. PhD thesis, Carnegie Mellon University, 1978.

Melvin E. Conway. Proposal for an UNCOL. *Commun. ACM*, 1(10):5–8, 1958.

Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004. ISBN 1-55860-699-8.

Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, 6(4):505–526, 1984a.

Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 111–116, New York, NY, USA, 1984b. ACM Press. ISBN 0-89791-139-3. doi: http://doi.acm.org/10.1145/502874.502885.

Joao Dias. *Automatically Generating the Back End of a Compiler Using Declarative Machine Descriptions*. PhD thesis, Harvard University, 2008.

Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. Beg - A Generator for Efficient Back Ends. In *PLDI*, pages 227–237, 1989.

Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. Effective compiler generation by architecture description. *SIGPLAN Not.*, 41(7):145–152, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1159974.1134671.

A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *EDTC '95: Proceedings of the 1995 European Conference on Design and Test*, pages 503–507, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7039-8.

Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler with C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991. ISBN 0-8053-2166-7.

Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/131080.131089.

Christopher Warwick Fraser. *Automatic generation of code generators*. PhD thesis, Yale University, 1977.

Mahadevan Ganapathi and Charles N. Fischer. Description-driven code generation using attribute grammars. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–119, New York, NY, USA, 1982. ACM Press. ISBN 0-89791-065-6. doi: http://doi.acm.org/10.1145/582153.582165.

Mahadevan Ganapathi and Charles N. Fischer. Affix grammar driven code generation. *ACM Trans. Program. Lang. Syst.*, 7(4):560–599, 1985. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/4472.4486.

Mahadevan Ganapathi, Charles N. Fischer, and John L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys*, 14(4):573–592, 1982.

R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 231–254, New York, NY, USA, 1978. ACM Press. doi: http://doi.acm.org/10.1145/512760.512785.

Robert Steven Glanville. *A machine independent algorithm for code generation and its use in retargetable compilers*. PhD thesis, University of California, Berkeley, 1977.

Susan L. Graham, Robert R. Henry, and Robert A. Schulman. An experiment in table driven code generation. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 32–43, New York, NY, USA, 1982. ACM Press. ISBN 0-89791-074-5. doi: http://doi.acm.org/10.1145/800230.806978.

Torbjorn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 341–352, 1992.

Michael Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 1–8, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-302-6. doi: http://doi.acm.org/10.1145/1128022.1128023.

Mary Hall, David A. Padua, and Keshav Pingali. Compiler research: The next 50 years. *Commun. ACM*, 52(2):60–67, 2009.

David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley, 1995. ISBN 0-8053-1670-1.

Roger Hoover and Kenneth Zadeck. Generating machine specific optimizing compilers. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–229, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237779.

Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Vols 1–3*, 2003.

Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C–: A portable assembly language that supports garbage collection. In *PPDP*, pages 1–28, 1999.

Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 304–314, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512566.

Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6):967–989, 2006. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1186632.1186633.

J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005. ISSN 0018-8646.

Peter B. Kessler. Discovering machine-specific code improvements. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 249–254, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-197-0. doi: http://doi.acm.org/10.1145/12276.13336.

B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production quality compiler compiler project. *Computer*, 13(8):38–49, 1980. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.1980.1653748.

Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

Christian Lindig and Norman Ramsey. Declarative composition of stack frames. In *Proc. of Compiler Construction'04*, pages 298–312, 2004.

Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS-II: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-8186-0805-6. doi: http://doi.acm.org/10.1145/36206.36194.

Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification For A New Family of RISC Processors*. Morgan Kaufmann Publishers, 1994.

W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, 1965. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/364995.365000.

Gordon E. Moore. *Cramming more components onto integrated circuits*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8.

J. Eliot B. Moss, Trek Palmer, Timothy Richards, II Edward K. Walters, and Charles C. Weems. CISL: a class-based machine description language for co-generation of compilers and simulators. *Int. J. Parallel Program.*, 33(2):231–246, 2005. ISSN 0885-7458. doi: http://dx.doi.org/10.1007/s10766-005-3587-1.

Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

Terence Parr. A Java template engine, November 2009. http://www.stringtemplate.org.

E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: an application of BURS theory. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 294–308, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: http://doi.acm.org/10.1145/73560.73586.

Todd Proebsting. Burg, iburg, wburg, gburg: so many trees to rewrite, so little time (invited talk). In *RULE '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, pages 53–54, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-606-4. doi: http://doi.acm.org/10.1145/570186.570191.

Todd A. Proebsting. BURS automata generation. *ACM Trans. Program. Lang. Syst.*, 17 (3):461–486, 1995. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/203095.203098.

Norman Ramsey and Jack W. Davidson. Specifying instruction semantics using CSDL (preliminary report). Technical report, Charlottesville, VA, USA, 1997.

Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. Technical report, Charlottesville, VA, USA, 1998.

Norman Ramsey and Mary F. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3): 492–524, May 1997.

Norman Ramsey, Jack W. Davidson, and Mary F. Fernandez. Design principles for machine-description languages. URL `http://www.eecs.harvard.edu/ nr/pubs/desprin.pdf`.

Timothy D. Richards, Edward K. Walters II, Trek Palmer, J. Eliot B. Moss, and Charles C. Weems. A unified framework for the automatic generation of system tools and components. Technical report, University of Massachusetts Amherst, 2007.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. ISBN 0137903952. URL `http://portal.acm.org/citation.cfm?id=773294`.

David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201737191.

Richard M. Stallman. Using and porting the GNU compiler collection (GCC). URL `http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html`.

William A. Wulf. PQCC: A machine-relative compiler technology. Technical report, Carnegie-Mellon University, 1980.